

Just Enough Pascal™

*To get you started programming
your Macintosh*

User's Guide

Credits

Authors

Walters & Associates
Chip Morrison and Joseph Walters

Product Manager

Diana Bury

Packaging and Production

Doreen Duplin

Quality Assurance

Paul Vetri

Additional Contributors

Barbara Clemens, David Greenes,
David Niguidula, Bryant Patten, Jim Rogers

Copyright © 1988 Symantec Corporation. All Rights Reserved.
135 South Road
Bedford, MA 01730, USA
(617) 275-4800

Symantec is a registered trademark and Just Enough Pascal and THINK's Lightspeed Pascal are trademarks of Symantec Corporation. "Lightspeed" is a registered trademark of Lightspeed, Inc., and is used with its permission.

The product names mentioned in this manual are the trademarks or registered trademarks of their manufacturers.

Contents

Introduction and Installation	1
The GridWalker Project	7
The JEP Instructions Window	15
 Assembly Stages	
Stage One: A Simple Program.....	23
Stage Two: Repeat Loops.....	31
Stage Three: Simple Animation.....	37
Stage Four: User-Defined Types.....	45
Stage Five: Functions.....	55
Stage Six: “Character” Records.....	63
Stage Seven: Simple Algorithms.....	69
Stage Eight: Random Numbers.....	77
Stage Nine: Arrays.....	85
Stage Ten: More About Arrays.....	95
Stage Eleven: Full Animation.....	103
Stage Twelve: The Palette.....	109
Stage Thirteen: The Main Event Loop.....	115
Stage Fourteen: Activating the Palette.....	123
Stage Fifteen: Activating the Grid.....	131
Stage Sixteen: Adding and Deleting Records.....	137
Stage Seventeen: The Menu Bar.....	143
Stage Eighteen: Adding Menus.....	149
Stage Nineteen: Adding a Strategy.....	153
Stage Twenty: Finishing Touches.....	159
 Technical Reference	
Starting at a Stage	185
Index	187
Licensing Agreement	191

|

|

Introduction and Installation

About JEP...

Just Enough Pascal (JEP) introduces you to the art of programming in Pascal on the Macintosh. And although JEP concentrates on the use of THINK's Lightspeed Pascal (THINK Pascal), you'll learn concepts that work with any Pascal development system.

JEP takes you step by step through the assembly of a completely functional Macintosh application using THINK Pascal. Along the way, you'll learn enough Pascal to begin creating your own Macintosh applications.

Who is JEP for?

JEP is for anyone who wants to use Pascal on the Macintosh. This includes anyone from the novice to the experienced Pascal programmer. For example:

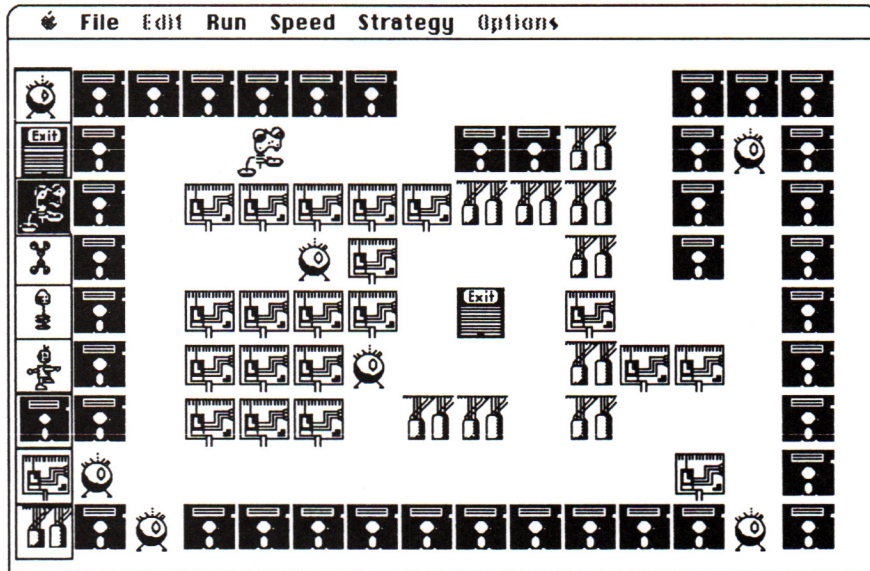
- If you have no programming experience with any computer language, JEP teaches you the fundamentals you need to get started.
- If you are familiar with Pascal but just want to improve your command of the language, you can hone your skills on the many programming challenges that JEP provides.
- If you are familiar with Pascal but have little or no experience programming on the Macintosh, JEP provides a good introduction to menu bars, event loops, and other special features of the Macintosh programming environment.
- If you are a student in a formal Pascal course, JEP helps you gain valuable experience of the kind you don't get in most classrooms.
- If you are teaching a formal Pascal course, you can use JEP as a supplement to your basic primer or even as the central focus for an introductory course.
- If you have general experience with Pascal, JEP introduces you to the specifics of THINK Pascal.

Basic Assumptions

JEP assumes that you are familiar with the basics of the Macintosh interface and its associated terminology, such as clicking and dragging. This guide defines some of these terms as it goes along, but if you're completely new to the Mac, you should read through your *Macintosh User's Guide* before tackling JEP.

The GridWalker Program

As you use JEP, you will be gradually putting together a computer program called **GridWalker**. In this program, small characters use a primitive sort of intelligence to navigate their way through complicated mazes. The main screen of the GridWalker program looks something like this:



The completed GridWalker program

Using the GridWalker program

GridWalker is a true Macintosh application. It conforms to the standard Macintosh interface guidelines, meaning you (and your users) can run it the same way you run any other Macintosh program. You select animated characters and icons from the palette, and using pull-down menus you can perform actions such as starting and stopping the animation. And because you're the programmer of GridWalker, you can customize it or even redesign it.

Assembling GridWalker

GridWalker is given to you as a kind of construction kit that comes with a collection of Pascal procedures and functions as well as a small library of icons and digitized sounds.

You assemble the GridWalker program one stage at a time. At the end of each stage, you will have a working computer program.

Learning Pascal with JEP

JEP illustrates an important principle of software development: programming proceeds from the ground up. You begin with a basic plan in mind, in this case, the overall design and structure of GridWalker. Then you get to work building and testing small pieces of the program, gradually assembling the parts into larger and larger components until finally the whole system is complete.

To get you started, JEP gives you a very simple program consisting of only a few Pascal instructions. By adding additional instructions, you assemble a slightly more elaborate version. Then you test the revised version and tinker with it to see how it works. Then you add some more instructions and test the results. This continues until the application is complete, or until you're satisfied with the results.

By the time you complete work on GridWalker, you will be skilled enough in Pascal to start building your own Macintosh applications, from the ground up.

Copy and Paste

Many introductory programming texts force you to type in all the examples. In the JEP environment, you don't have to do a lot of tedious typing. Instead JEP makes it possible for you to **copy** and **paste** lines of the code into GridWalker as you create it, quickly and precisely. This allows you to concentrate on the important aspects of learning to program, rather than on the accuracy of your typing skills.

Assembly—Explanation—Tinkering

The process of assembling GridWalker is broken down into twenty carefully-designed stages. Each stage begins with the **Assembly** step. JEP provides you with new Pascal program code and tells you where to paste it in GridWalker. Then you run the program to see how the new version differs from the previous version.

Each stage also focuses on particular aspects of Pascal, such as *variables*, *loops*, *conditionals* and *arrays*. To help you grasp these concepts more fully, each stage contains an **Explanation** section that discusses the new concepts in detail. This way you learn the Pascal language in the context of hands-on programming.

Finally, a **Tinkering** section, at the end of each stage, suggests experiments you can conduct with each new version of the program. You'll be able to run the program on your Macintosh, make changes to it, and use the special features of THINK Pascal to look deep inside the program while it's running. Some of the features you'll be able to use include the Observe and Instant windows, LightsBug, Stop Signs, and the Step and Trace commands.

Troubleshooting and Cross referencing

JEP's **Troubleshooting** section helps you find and diagnose errors when things don't go exactly as planned. **Cross references** are available throughout—they link certain key words (shown in **boldface**) to a Pascal reference section. This helps you quickly look up definitions of unfamiliar terms.

The JEP Disk and User's Guide

You'll need to have THINK Pascal (purchased separately) installed on your system. Everything else you need to assemble the GridWalker program is provided on your JEP disk. It contains the code you need (which is always available to you in the JEP Instructions window), complete explanations of how the program works, suggestions for experimenting with the program, and quick access to key Pascal terms.

This *JEP User's Guide* contains everything you need to know to use JEP. It contains two parts. Besides introducing you to JEP, this first part describes the steps you must complete to install THINK Pascal and the software components of JEP. It also explains how to use the JEP Instructions window.

The second part explains how the program works at each stage in the assembly process. This part gives an overview of the Assembly and Tinkering sections and duplicates the on-line Explanation sections along with additional graphics. You can read the Explanation either on the screen or in the manual. In this way the manual is a handy way to learn more about the program even when you aren't working at your Macintosh.

The Appendix consists of a Technical Reference, which briefly describes all the icons, constants, variables, functions, and procedures used in GridWalker, and instructions for starting the assembly at a particular stage.

Register Your Copy of JEP

Your JEP package contains a Product Registration Card that you should complete and return immediately to Symantec. By registering your purchase, you are assured of receiving news of future upgrades. Furthermore, this ensures that you receive prompt service should you ever have technical questions about JEP.

Copy the JEP Disk

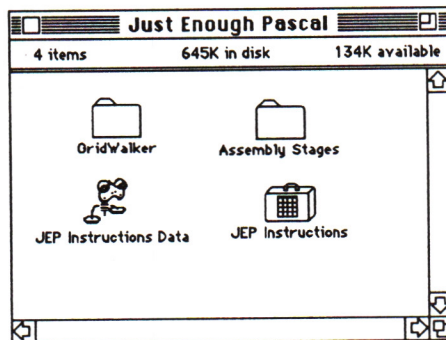
Experienced computer users know the importance of backing up their product disks. Before you do anything else, make a copy of your JEP disk. Store the original disk in a safe place and use only your copy from here on. All future references in this guide to the *JEP Disk* refer to your copy.

If you need help making a backup copy, please refer to your *Macintosh User's Guide* for instructions on copying floppy disks.

Verify the JEP Disk

Next, verify that the JEP disk contains all of the necessary components.

1. Insert your JEP disk into one of your floppy disk drives. The disk's icon then appears on your Macintosh desktop (the visual representation of the Macintosh system software component known as the **Finder**).
2. Double click on the JEP disk icon to open its window. You should see four icons:



- **GridWalker** folder—contains the materials you will need to assemble the GridWalker program, including the documents GridWalker Project and Main Program, the resource file JEP Resources.rsrc, and eight other Pascal documents.
- **The Assembly Stages** folder—contains 21 folders, named *Assembly Core* and *Stage 1 Complete* through *Stage 20 Complete*. Each folder contains the source code as it stands at the completion of that stage in the assembly process. See the Appendix for instructions on how to use the contents of this folder to begin the assembly process at any given stage.
- **The JEP Instructions** desk accessory—this desk accessory contains the instructions for assembling GridWalker.
- **JEP Instructions Data**—contains the data used by the JEP Instructions DA.

Installing THINK Pascal and JEP on a Hard Disk

Start the Finder. If you've installed THINK Pascal on you hard disk, please skip ahead to the instructions for installing JEP.

Installing THINK Pascal

1. Create a new folder on your hard disk called THINK Pascal Folder.
2. Open your Pascal application disk and locate three documents: the THINK Pascal application and two libraries, Runtime.lib and Interface.lib. Drag all three into the new folder. You can also copy the other documents onto your hard disk as well, but these three are all you need when using JEP.

Installing JEP

1. Use the Font/DA Mover™ to install the **JEP Instructions** desk accessory into the System file in your System folder.

You will find the Font/DA Mover on one of the utilities disks you received with your Macintosh. The *Macintosh Utilities User's Guide* explains how to use Font/DA Mover. Suitcase™ and Font/DA Juggler™ can also be used to install JEP Instructions.

2. Copy the **JEP Instructions Data** document from your JEP disk into the System folder on your hard disk.
3. Copy the **GridWalker** folder to your hard disk.
4. If you have enough space, copy the **Assembly Stages** folder to your hard disk. Otherwise, save this folder on a backup floppy disk.

Installing THINK Pascal and JEP on 800K Disks

These instructions for installing THINK Pascal are different from those in the *User's Manual*. The configuration described here is more practical for running JEP. Start the Finder.

1. Format an 800K disk to serve as a startup disk.
2. Copy your System folder to this disk.
3. Copy the **JEP Instructions Data** document from the JEP disk into your System folder. If you need room, use the Font/DA mover to remove fonts and desk accessories from the System.
4. Use the Font/DA Mover to install the **JEP Instructions** desk accessory into your System file in your System folder.

You will find the Font/DA Mover on one of the utilities disks you received with your Macintosh. The *Macintosh Utilities User's Guide* explains how to use Font/DA Mover. Suitcase™ and Font/DA Juggler™ can also be used to install JEP Instructions.

5. Format a second 800K disk to serve as your Pascal work disk.
6. Create a new folder called THINK Pascal Folder on this disk .
7. Open your Pascal application disk and locate three documents: the THINK Pascal application and two libraries, Runtime.lib and Interface.lib. Copy all three into the new folder.
8. Copy the **GridWalker** folder from the JEP disk onto the THINK Pascal work disk.

The GridWalker Project

Now that you've installed all the necessary components, you're prepared to start assembling the GridWalker program. Before you actually begin, however, you need to know a little about how THINK Pascal works.

Working With THINK Pascal

The components of THINK Pascal work together as a sort of team to help you develop your program. You use different ones at different times, depending on the task:

Editor

The editor in THINK Pascal works like a word processor—you use it to type in new code and to insert, erase, copy, and paste when needed. Whenever you look at the code in your program, you are using the editor.

Compiler

Pascal code is similar to English. Unfortunately, the Macintosh does not understand English. The compiler translates the Pascal code (called the **source code**) into a language the Macintosh can understand (called the **object code**).

Linker

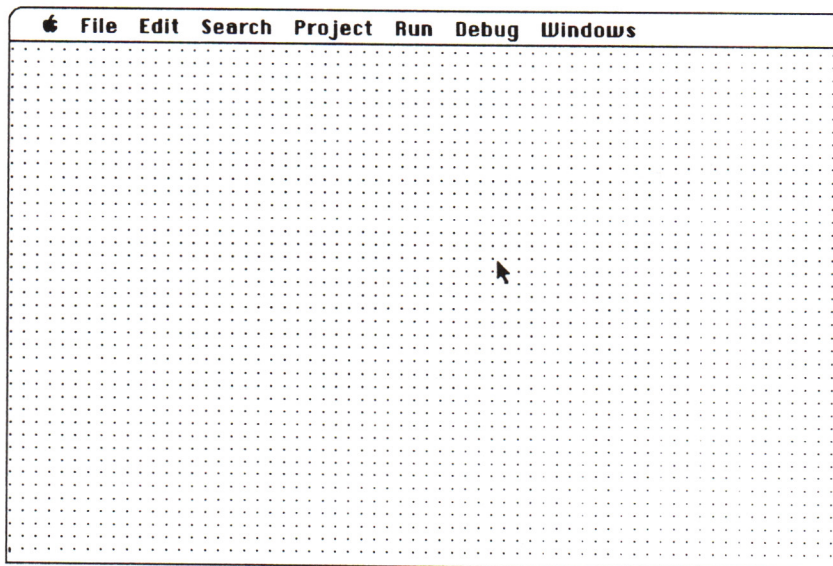
This component links the various parts of the program together.

Debugging Tools

A large part of program development involves isolating and correcting bugs in the program code. The Debugging Tools in THINK Pascal help you with this task. Some of the tools you have available include the Observe and Instant windows, LightsBug, the Step and Trace commands, and Stop Signs.

Launching THINK Pascal

In Macintosh terminology, to **launch** an application means to start it. To launch THINK Pascal, double-click on its icon on the desktop. You'll see a blank screen with the menu bar at the top.



The opening screen for THINK Pascal.

Your screen may have a different background pattern—you can change this pattern using the Control Panel desk accessory.

GridWalker Project Document

When you write a program using THINK Pascal, you actually create *several* documents. These documents contain information necessary to the program. You will also create a **project document**—this is the manager that keeps track of all the related sources of information. All documents related to the GridWalker Project are saved in the GridWalker folder.

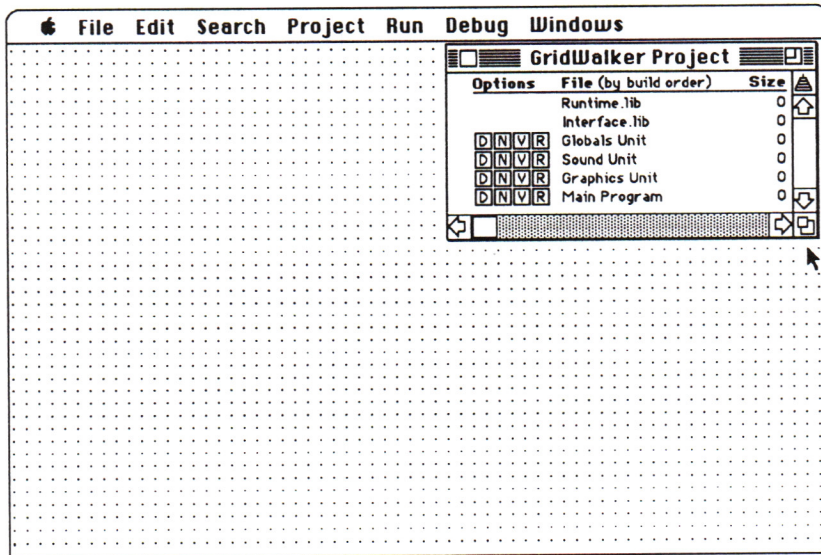
Macintosh Note

In the Macintosh environment, files are called **documents**. Disks and other volumes can be organized in separate directories called folders. While working at the desktop, you can manipulate documents and folders by renaming them, copying their contents, erasing them, opening their windows, and so forth.

Whenever you write a new program, you start by creating a new project document. You then write the program itself in one or more separate documents, which you add to the project. To make changes to a program, you first open the project document and use it to open the source documents. All of this information—the source documents and the project document—is called the **project**.

To get off to a quick start assembling GridWalker, you'll use the project document located inside the GridWalker folder called **GridWalker Project**. To open this project:

1. Pull down the Project menu and choose **Open Project...** A standard Macintosh file dialog box appears, listing all the folders and project documents on the current disk.
If the GridWalker folder isn't on your current disk, use the **Drive** button to switch to a different drive.
2. Select the GridWalker folder by clicking on its name, then click **Open**.
3. Locate the document named GridWalker Project, click to select it, and then click **Open**. This opens the GridWalker Project window. Drag the size box at the lower right of the window to make the Project window smaller.



The GridWalker Project window

GridWalker Project Window

Let's take a moment to discuss the contents of this window.

Libraries

These documents are listed under the File heading and can be identified by the **.lib** suffix to their names. Library documents contain Pascal object code that has already been translated, by the compiler, from the original source code. Runtime.lib and Interface.lib are automatically included in *every* THINK Pascal project. They contain basic Pascal and Macintosh commands you can use in your own programs. Other libraries are included with THINK Pascal, but are not used here.

Units

Because GridWalker is a fairly large program, it has been divided into several smaller pieces called **units**. Working with these units makes it easier to find things in the program

and correct errors. The GridWalker Project window lists three units—the Globals Unit, Sound Unit, Graphics Unit—and a main program (called Main Program). These four files were created specifically for GridWalker.

Build Order

The GridWalker units are designed to function in a particular order called the **build order**. THINK Pascal has no way of knowing what the build order should be, so it is up to you to make sure it is correct. The correct build order for the GridWalker Project is shown in the preceding sample screen and is set this way in your Project Document.

Options

The letters **D**, **N**, **V**, and **R** in the small boxes next to the unit names are options for using the Debugging Tools. A letter enclosed in a box indicates the option is enabled. Click on these letters to enable or disable the options. See the THINK Pascal *User's Manual* for details.

Size

This column indicates the size of the library or unit document.

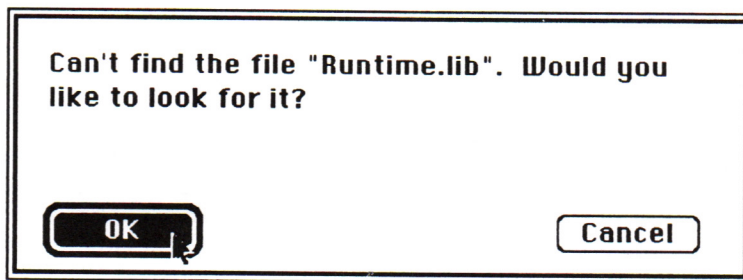
Running the Project

The GridWalker project is completely functional, so let's take it out for a test drive just to see what it can do.

To run GridWalker, choose **Go** from the Run menu.

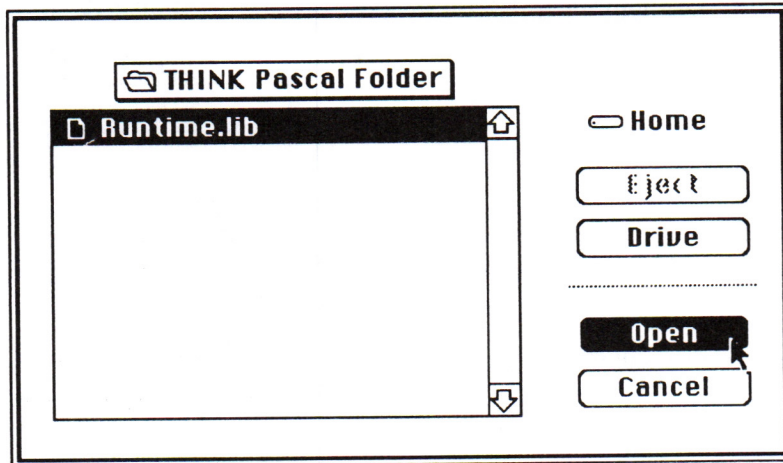
Before THINK Pascal can actually run GridWalker, it must locate all the required documents, load them into memory, link them together, and compile them. Only then can it actually run the program. As THINK Pascal performs each step, it displays messages that tell you what is happening. Likewise, it displays an error message if it determines that something might be wrong.

Because this is the first time you have run GridWalker, THINK Pascal has no way of knowing where the documents in it are located. It displays a dialog box like the one that follows asking if you want to look for the first library.

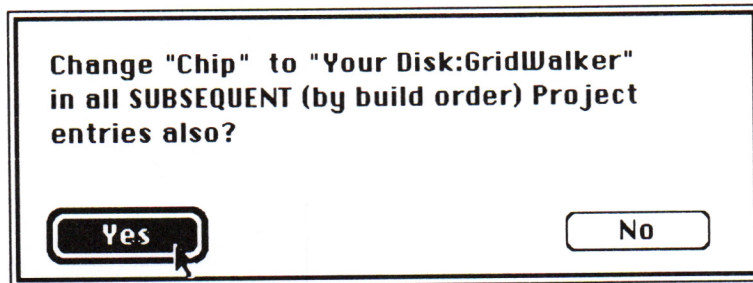


Click **OK**.

Next you see a dialog box similar to the one shown below. Locate the folder called THINK Pascal Folder (you may have to change drives). The document **Runtime.lib** is in that folder.



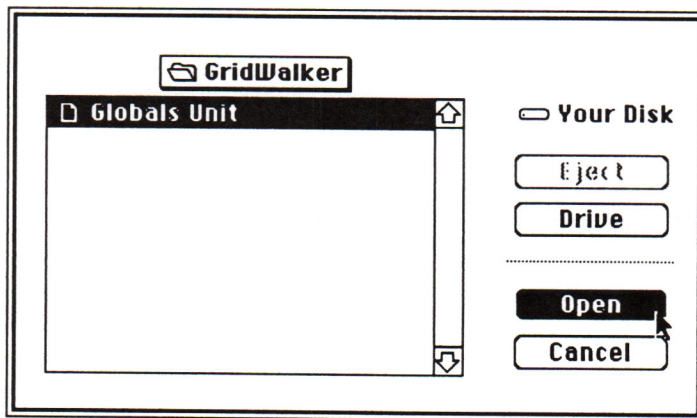
Click **Open**. The next dialog you see (below) asks whether the remaining documents in the build order are also located inside the same folder as the first one—that is, in the THINK Pascal Folder. In this case, the next file **Interface.lib** is in the same folder.



Click **Yes**. This changes the record of the location of these documents kept by the project document.

Next, THINK Pascal tries to compile the first unit of the project, the Globals unit. It doesn't find this unit in the folder with the libraries, so it displays the message "Can't find the file Globals Unit. Would you like to look for it?" Again, click **OK**.

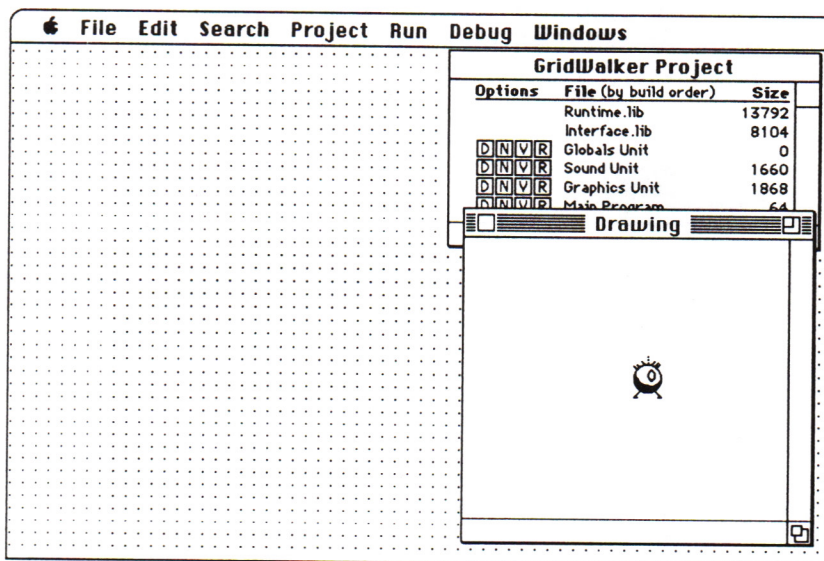
And again, the file selection box is displayed. This time, locate the GridWalker folder. The Globals Unit is in that folder and its name is highlighted. Click **Open**.



Finally, you will once again see the message “Change ... in all SUBSEQUENT (by build order) Project entries also?” Click **Yes**, because the remaining units in the project are also located in this folder.

THINK Pascal compiles the remaining units and then runs the program. It also records the locations of these documents in the Project document, so you don't have to repeat the preceding steps every time you run the program.

After THINK Pascal runs GridWalker, you see the results (or **output**) of the program on your screen. In this case, the Drawing window opens automatically and a small drawing of a bomb appears inside, as the next figure shows:



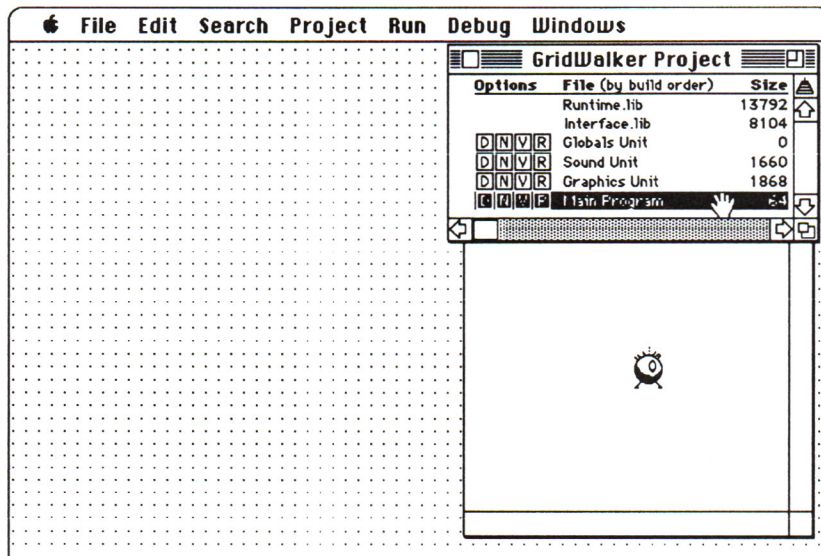
The display after running the project

Opening the Main Program

Every THINK Pascal project has a main program. This is where the program starts. It's also a good place to begin when figuring out how the program works. It's always the last document in the project window and it need not be called "Main Program."

To open the main program in the GridWalker Project:

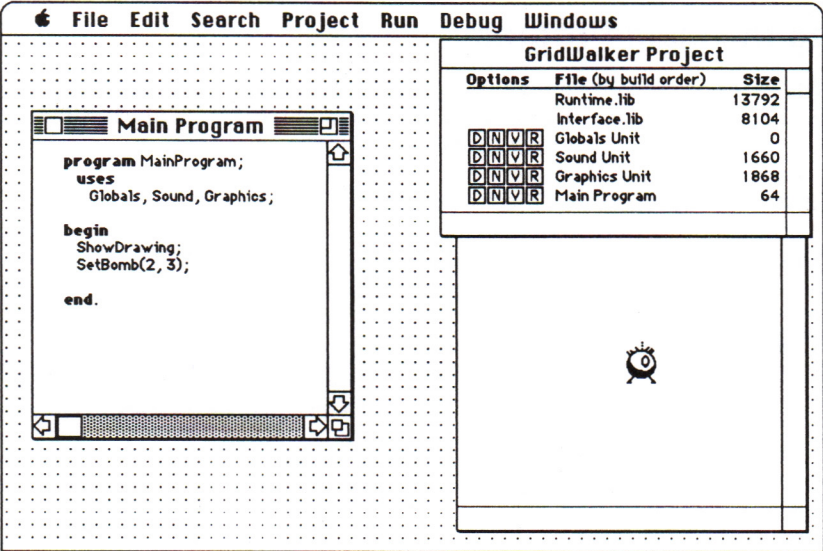
1. Click on the GridWalker Project window and move the pointer over the words *Main Program* in the Project window. You'll see the pointer turn into a small hand.



Opening the Main Program window

2. Double-click and a new window opens with the main program inside.

This type of window (shown on the next page) is called an **edit window**. It displays the program's source code. You can make changes to the source code in this window. You can also resize the window and move it around on your screen.



The Main Program window

Building on the Main Program

The main program currently performs a relatively simple task: it opens the Drawing window and draws the bomb icon at a specified location in that window. While this might not seem an impressive achievement, it does provide you with a good starting point. As you work through the assembly process, you'll build on the main program until you have your own sophisticated Macintosh application complete with windows, menus, a palette, sounds, and animation.

Before you can begin the assembly process, however, there is one more JEP component you need to know about—the **JEP Instructions** window.

The JEP Instructions Window

Introduction

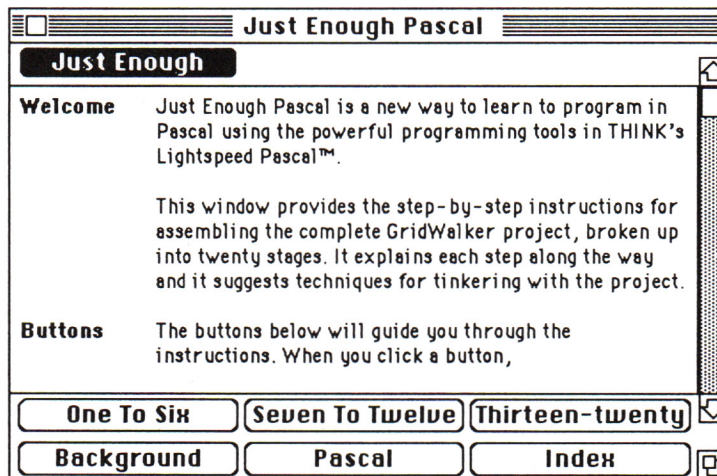
The JEP Instructions window is like your personal guide, and more. It provides the source code you will need as you assemble the project. It explains how the program works at each stage. It suggests ways to tinker with your program. And it challenges you to make changes.

The JEP Instructions window is unique to Just Enough Pascal. It's a sophisticated indexing system that helps you *navigate* your way through an extensive amount of information.

You can click buttons, choose menu commands, or follow electronic cross references. This section explains how to use this system.

Learning to Navigate

To open the window, choose **JEP Instructions** from the Apple menu. If **JEP Instructions** isn't listed in the **Apple** menu, the JEP Instructions desk accessory has not been properly installed. Go back to the *Introduction and Installation* section earlier in this guide and install the desk accessory.

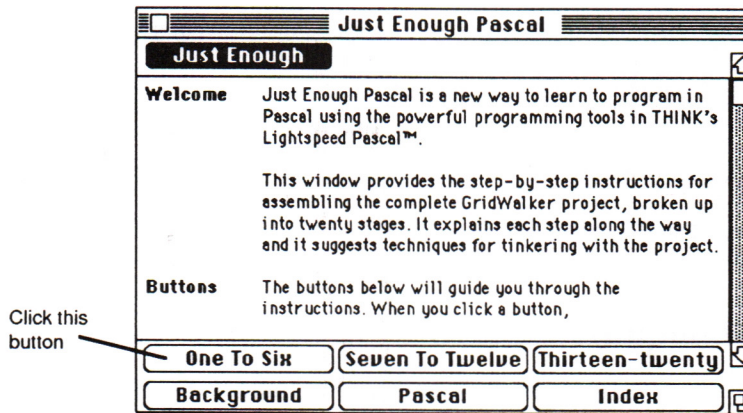


When you first open the JEP Instructions window, there is one button (**Just Enough**) at the top of the screen. It is highlighted to show that this is the screen you are currently looking at. The six buttons at the bottom take you to other screens. The first three (**One To Six**, **Seven to Twelve**, **Thirteen To Twenty**) contain instructions for the twenty assembly stages.

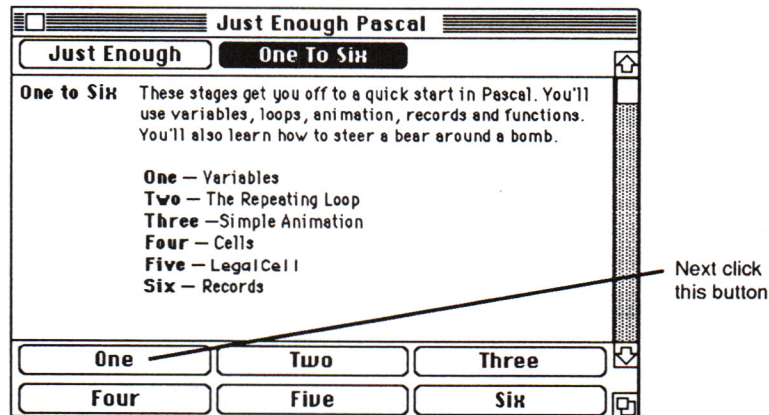
Background contains background information, **Pascal** contains the Pascal reference section, and **Index** gives you an electronic index of key terms.

Getting to the Assembly Screen for Stage One

To illustrate how the buttons work, let's use them to locate the Assembly screen for Stage One. This is where you will find instructions for beginning the assembly process.



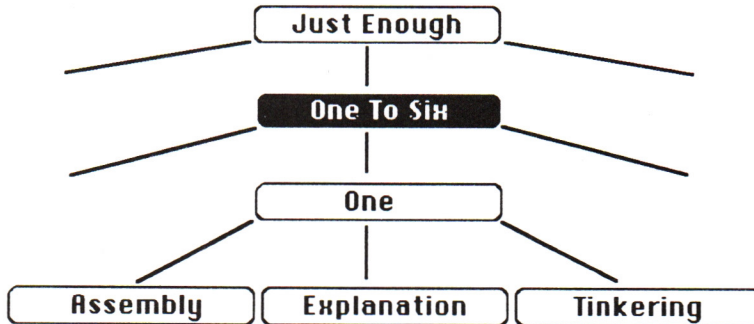
First, click on the button labeled **One To Six**. Your window should now look something like this:



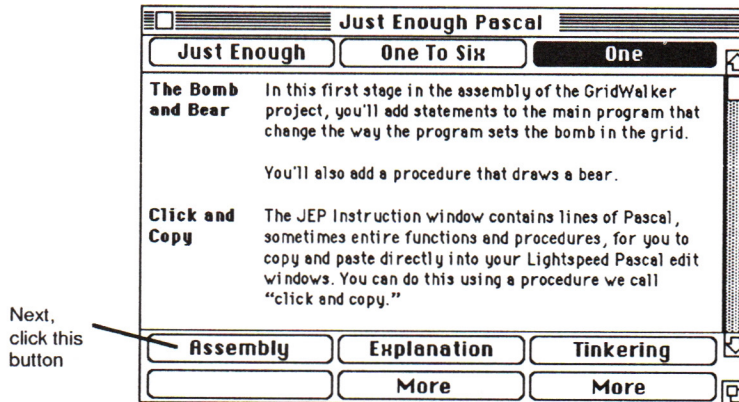
The **One To Six** button has moved to the top of the window and is now highlighted because this is the screen you are looking at. A new set of buttons appears at the bottom of the window.

The hierarchy of buttons

These buttons are organized in a hierarchy like a tree. The button **Just Enough** is at the top of this tree and it has six buttons under it. When you clicked **One To Six**, you moved down one level along a branch of the tree. At this point button **One** is below you and the button **Just Enough** is above. In a moment, you will see the buttons below **One**.



To continue down along a branch, click the button labeled **One**, near the bottom of the window. The JEP Instructions window should now look like this:

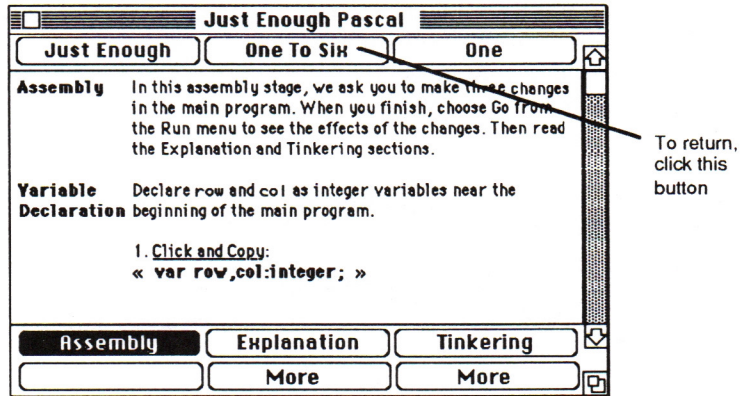


Once again, the button you selected moves to the top of the window and is highlighted. A new set of buttons appears at the bottom of the window.

This screen contains the opening discussion for Stage One. Our objective here is to get to the Assembly instructions.

Just Enough Pascal

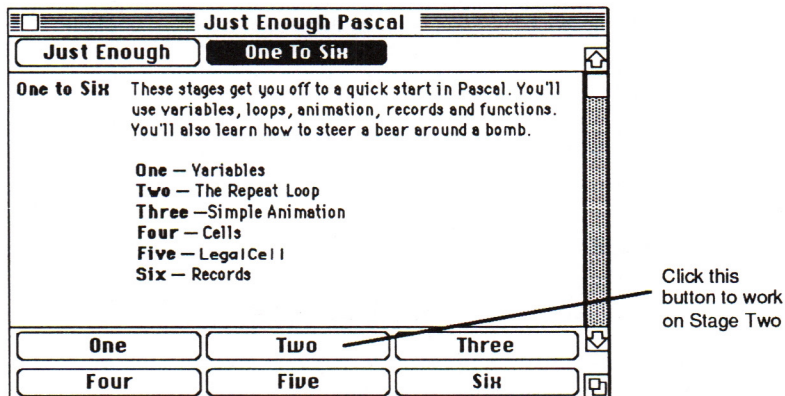
Click the **Assembly** button and your window changes to look like this:



This time the **Assembly** button is highlighted but it did not move to the top of the window. This means you have come to the end of this particular branch. You can read the Assembly Instructions by scrolling the text through the window.

The **Explanation** button on this screen takes you to an explanation for Stage One. The **Tinkering** button reveals directions for changing the source code in Stage One and offers challenges for personalizing the program. Both the Tinkering and Explanation sections are supplemented with additional screens represented by the **More** buttons below them.

To move back up the tree, click buttons in the top row of the window. Click **Just Enough** to return quickly to the opening screen. Or click on **One To Six** to jump up two levels:



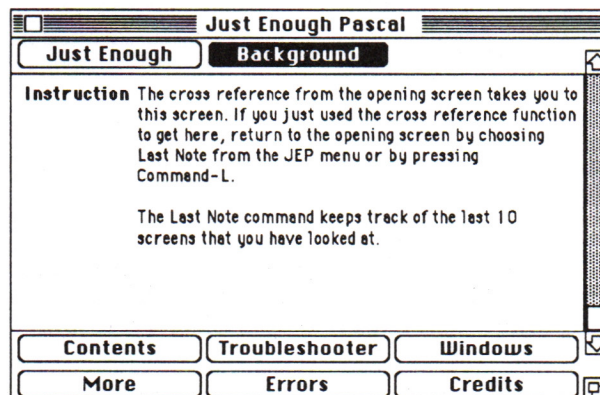
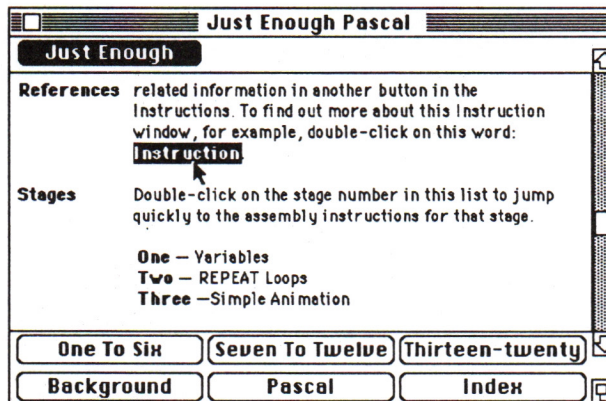
When you do, the **One To Six** button is highlighted. From this screen, you can go on to the Assembly instructions for Stage Two by clicking the button labeled **Two**.

Other Navigation Techniques

Clicking buttons is only one way of moving from one screen to another.

Using Cross References

Any term that appears on the screen in boldface has a **cross reference**. That means it is electronically connected to a related discussion in the Pascal reference section or somewhere else in the Instructions window. To follow this connection, double-click on the boldfaced word, or select it and choose **Cross Ref.** from the JEP menu.

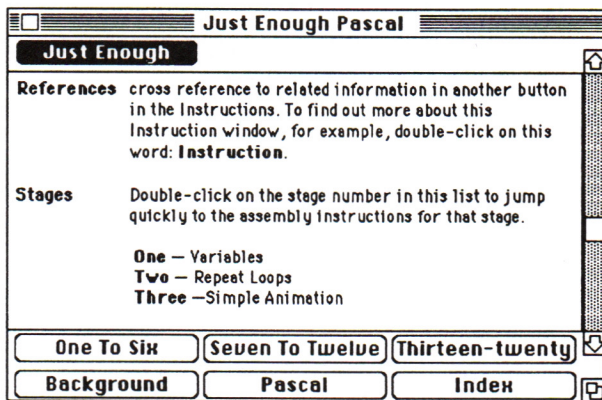
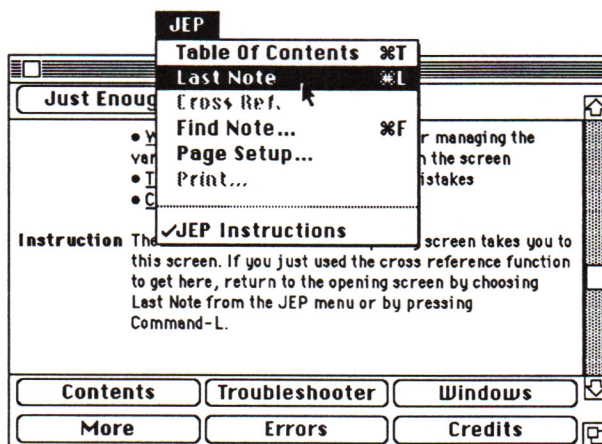


For example, return to the Just Enough screen and scroll down until you find the word **Instruction** in bold. Double-click on it. The screen containing the related information appears automatically. The buttons have also changed—you are now reading text in the **Background** button.

Navigating Backwards

Because a cross reference can take you almost anywhere in the Instructions, you need a quick way to get back to where you were. The **Last Note** command on the JEP menu does just that.

Select **Last Note** from the JEP menu or press Command-L. This takes you immediately to the last screen you looked at.



JEP always keeps track of the last 10 screens you viewed.

Continuing

Every screen in the twenty assembly stages ends with the heading **Continue**. Here you'll find a cross reference to the screen that comes next in the logical order. For example, at the end of

the **Tinkering** section in Stage One, the cross reference in the Continue section takes you directly to the beginning of **Stage Two**.

Index

To find a topic, click the button **Index**. Then scroll through the alphabetical list of terms and click the one you want. This cross reference will take you directly to a discussion of that topic.

Summary of the Instructions Menu

JEP	
Table Of Contents	%T
Last Note	%L
Cross Ref.	
Find Note...	%F
Page Setup...	
Print...	
.....	
✓JEP Instructions	

Table of Contents Command

Choose this command to display a Table of Contents of the buttons in the form of an outline. Go directly to any section by clicking on the button name.

Last Note Command

Choose this command to view the last screen you looked at. JEP remembers the last ten screens.

Cross Ref. Command

You'll often see words displayed in the JEP Instructions window in **boldface**. This indicates that the word is cross referenced to another topic of related information. For example, many technical terms have cross references to the Pascal reference section.

To access the cross reference, click on the boldfaced term and then choose the **Cross Ref.** command, or simply double-click the term. The related information appears on your screen instantly. After reviewing the cross reference, you can return directly to your starting point by choosing the **Last Note** command (see the preceding description).

Find Note... Command

This command helps you quickly locate any term in the JEP Instructions system that is cross referenced. When you choose the command, a dialog box appears in which you can enter up to 19 characters. Click the **Find Word** button in the dialog box and JEP displays that portion of the instructions in the window. Repeat the process to find the next occurrence of the word.

Page Setup... and Print Commands

You can create your own notes by typing text in empty buttons just like you type text into a word processor. Use the two commands, **Page Setup** and **Print**, to print these notes.

Starting the Assembly

It's time to get to work. Assembling the entire GridWalker project can take several hours. Don't try to do the entire project in one sitting. Instead, plan to complete it in several different sessions, allowing yourself adequate time to work through all of the instructions and explanations. You'll have a fully running program at the end of every stage, so you can stop at any point and then easily resume at a later time by simply opening your project and picking up where you left off.

Also allow yourself time to work through JEP's Tinkering sections. Although you can learn a great deal by simply completing the project as described, you'll develop your skills and understanding much faster by experimenting on your own. You'll make mistakes, of course, but that is all part of the learning process. JEP keeps you on course. When you're finished, you'll be ready to create your own programs.

As you work through each stage, follow the Assembly instructions first. Then read the Explanation and work on the Tinkering. Try your hand at the challenges. Here are some logical break points in the assembly process:

Stages One to Six: Characters, Cells, and Animation

Stages Seven and Eight: Strategies

Stages Nine to Eleven: Arrays and Refined Animation

Stages Twelve to Sixteen: The User Interface and Palette

Stages Seventeen and Eighteen: Menus

Stages Nineteen and Twenty: Finishing Touches

If you haven't completed all of the following steps, please complete them now:

1. Launch THINK Pascal.
2. Open the GridWalker Project document.
3. Run the GridWalker project by choosing **Go** from the Run menu.
4. Open the JEP Instructions window by choosing **JEP** from the Apple menu.
5. When the JEP Instructions window appears, click **One To Six**.
6. When the next window appears, click **One**.

You are now back to the opening discussion for Stage One of the GridWalker project. Follow the instructions now on your screen to begin assembly.

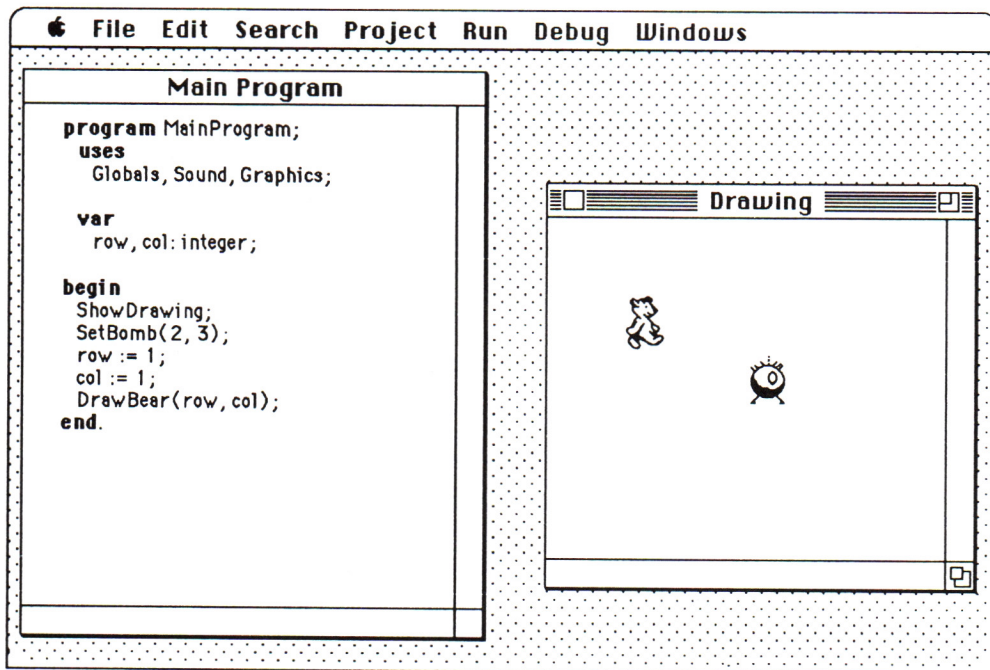
Good Luck!

Stage One

A Simple Program

The Bomb and Bear

When you first ran the GridWalker program, it simply drew a bomb on the screen. In this stage you introduce a bear into the picture. The bomb and the bear don't have anything to do with each other—yet.



The project after the assembly in Stage One

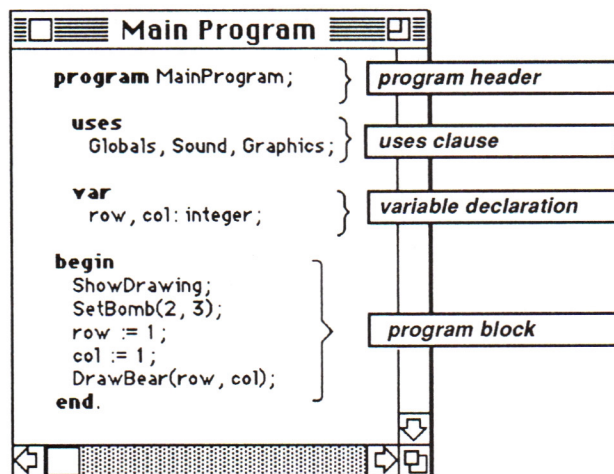
Assembly

In the assembly portion of Stage One you add three new parts to the main program:

- a variable declaration
- two assignment statements
- a call to the `DrawBear` procedure

Explanation Anatomy of a Program

Let's take a close look at the program in the Main Program window.



Like all Pascal programs, our program begins with a **program header**. This statement marks the very beginning of the program and gives it a name.

The **uses clause** links the main program to other units in the project. The **variable declaration** section that follows creates variables for the program. Both sections are discussed in more detail later.

Finally, the **program block** makes up the remainder of the program. It contains statements surrounded by the keywords **begin** and **end**.

begin

Words like **begin**, called keywords, serve a special purpose. For example, the keyword **begin** marks the beginning of a block of statements. In THINK Pascal all keywords are automatically printed in bold.

end

The end of the block is marked by the keyword **end**. In this particular case, **end** also marks the very end of the program, so it is followed by a period. You will never add statements below this line of the program.

The lines that fall between the keywords **begin** and **end** are **statements**. Statements must be separated from each other by semicolons.

Procedures

Take a look at the first statement in the program block. Notice that it is a single word:

```
ShowDrawing;
```

This statement is a **procedure call**; it means "Execute the procedure ShowDrawing." A **procedure** is a separate piece of code that performs a specific task. The procedure ShowDrawing, for example, opens the Drawing window.

When the program gets to this procedure call, it must know that it can find the definition of ShowDrawing in the library Runtime.lib. Therefore, Runtime.lib must come *before* Main Program in the build order. If the compiler does not find the procedure definition, it alerts you with an error message.

The actual code for the ShowDrawing procedure is located in the library Runtime.lib. Look at the GridWalker Project window and find this library listed at the top of the list of documents that are included in the project.

THINK Pascal Note

To make the Drawing window larger or smaller, resize it before you run your program. Later, you will learn to set the size of the Drawing window with a statement in the program itself.

SetBomb

The next statement in the program is a second procedure call:

```
SetBomb (2, 3) ;
```

This procedure draws a picture of the bomb in the drawing window. The two integers in parentheses, the **arguments** or **parameters**, tell the procedure precisely where to place the bomb.

The procedure `SetBomb` is defined in the Graphics unit. We will examine this unit in detail in Stage Three.

Variables

The next two statements in the program use variables for the first time. A **variable** represents a location in the computer's memory. Think of it as a box you store something in.

In Pascal you must **declare** each variable before you can use it, by specifying what **type** of variable it is. A variable of a particular type can only store values of that type. For example, a variable of type `integer` can store only **integers** (positive and negative whole numbers), not letters or numbers with decimal parts.

Find the variable declaration part of the main program:

```
var
    row, col: integer;
```

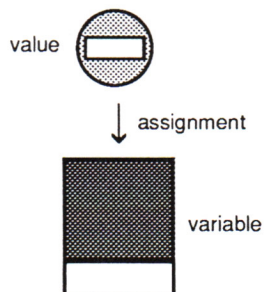
This variable declaration section creates two variables, `row` and `col`, and specifies that both will be of type `integer`. The keyword **var** marks the beginning of the variable declaration list.



Integers are only one type of variable in Pascal. Other types include **arrays**, **booleans**, **records** and several others. Many of these types are described in later stages of JEP.

Assigning values to variables

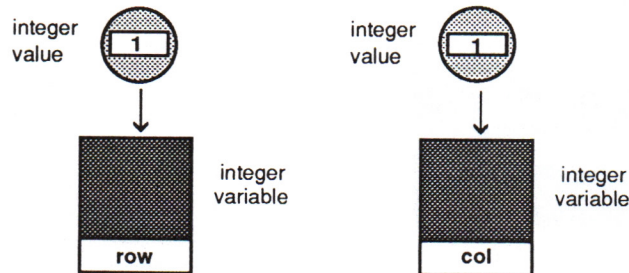
The thing you store in the variable box is called a **value**. Putting a value in the box is called "assigning a value to a variable."



Now find the following lines in the program block.

```
row := 1;
col := 1;
```

Both lines are **assignment statements**. They assign the value 1 to the integer variables `row` and `col`. After these assignments, the integer value 1 is stored in the two variables called “row” and “col”.



After this assignment is made, if we could peek inside these variable boxes, we would see the values.



THINK Pascal provides special tools for examining the values of variables. We'll show you how to use them later.

Pascal Note

The symbol `:=` is pronounced “gets.” The first assignment statement therefore reads “The variable `row` gets the value 1,” or simply “`row` gets 1.” In an assignment statement, a variable always appears on the left of the `:=` symbol and an **expression** appears on the right. We will consider expressions in more detail in the next stage.

Passing variables to procedures

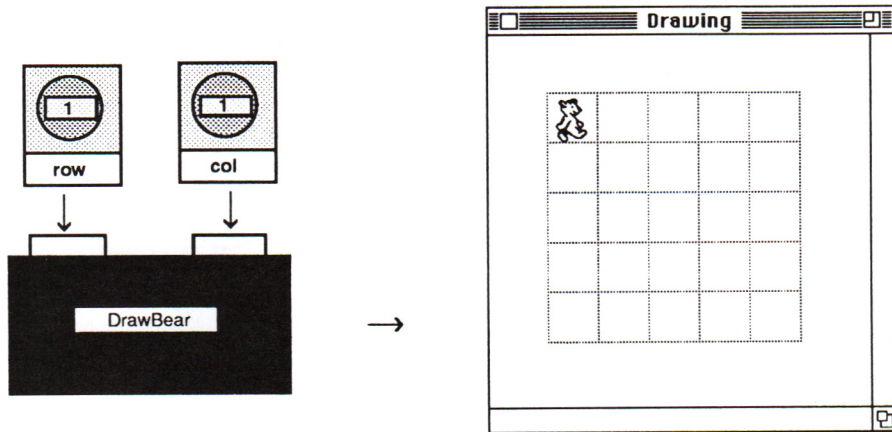
Next, look at the statement that passes the contents of our two variables to a procedure:

```
DrawBear(row,col);
```

`DrawBear` (also defined in the Graphics unit) takes two integer variables as parameters, just like `SetBomb`. In this statement, however, we are passing those integers in the form of the

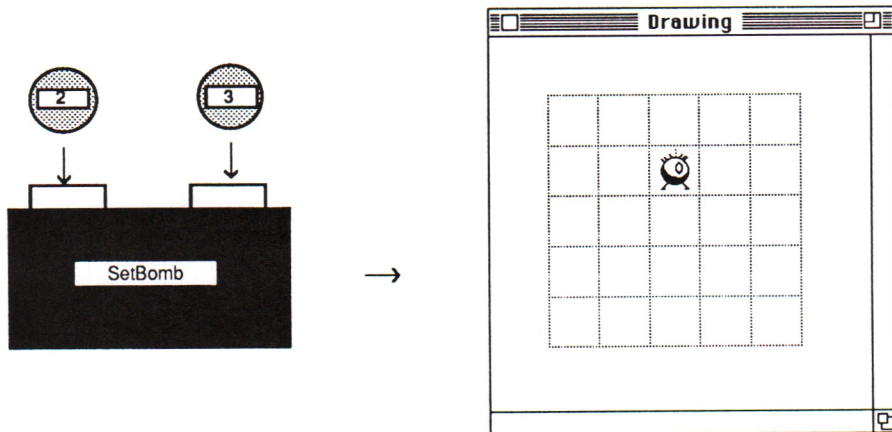
Just Enough Pascal

variables `row` and `col`. When this statement is executed, the current values stored in `row` and `col` are passed to `DrawBear`, which draws a picture of the bear in the specified row and column of the grid.



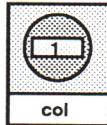
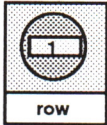
In the statement `DrawBear (row, col)`, the parameters to the procedure are variables. Because other program statements can change the values for the variables `row` and `col`, the single statement `DrawBear (row, col)` can actually draw the bear in different positions. We'll show you how to take advantage of these variables in the next assembly stage.

In the statement `SetBomb (2, 3)`, on the other hand, the parameters are values.

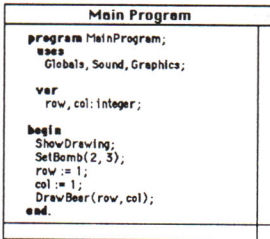


What is a Computer Program?

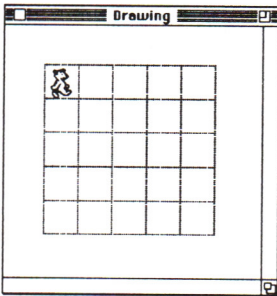
Now that we have examined this program in detail, let's step back and consider what we have been looking at. All programs, from our simple GridWalker program to large, complex applications, involve three basic components.



First, there is the data, consisting of variables and their values. Much of the work a program does involves creating or reading data, making calculations, and storing the results.



The work that a program does is accomplished with the program statements, or “code”. These statements are organized in a particular sequence, and executed one line at a time.



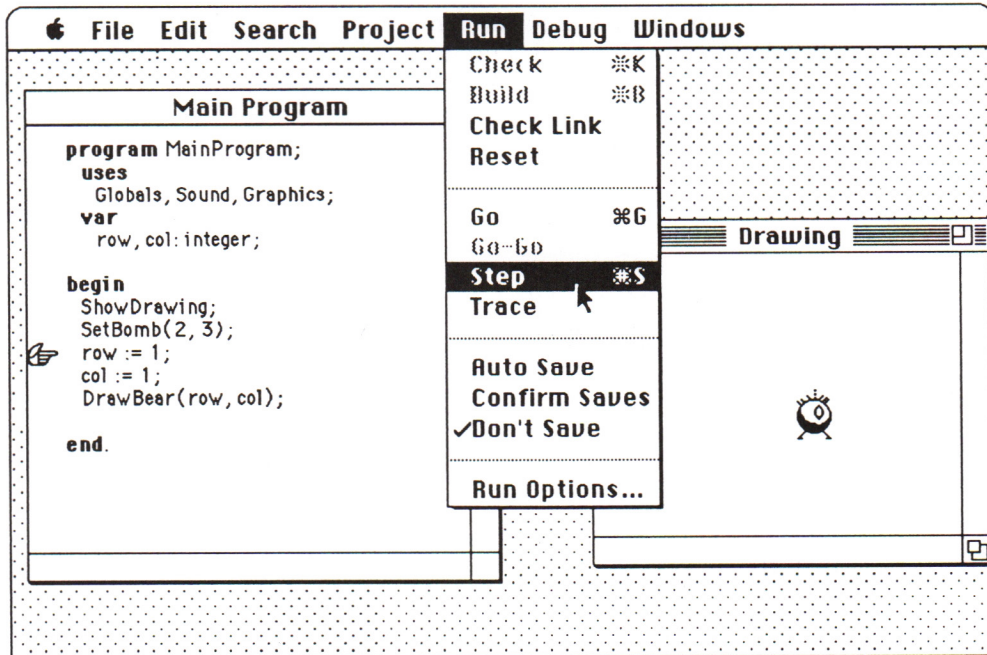
The results of the program statements, the output of the program, are the visible (and sometimes audible) effects of what happens when the code interacts with the data.

As GridWalker becomes more complicated, this basic structure will not change. The data will become more complex, the code will grow to several hundred lines, and the output of the program will become rich and varied. But GridWalker will continue to operate just as it does at this basic level: code acting on data to produce output—one step at a time.

Tinkering

The Step Command

In Stage One, the on-disk Tinkering section of the JEP Instructions tells you how to use the **Step** command in THINK Pascal. This command executes your program (carries out the instructions) one statement at a time.



Using the Step command with the main program

Stage Two

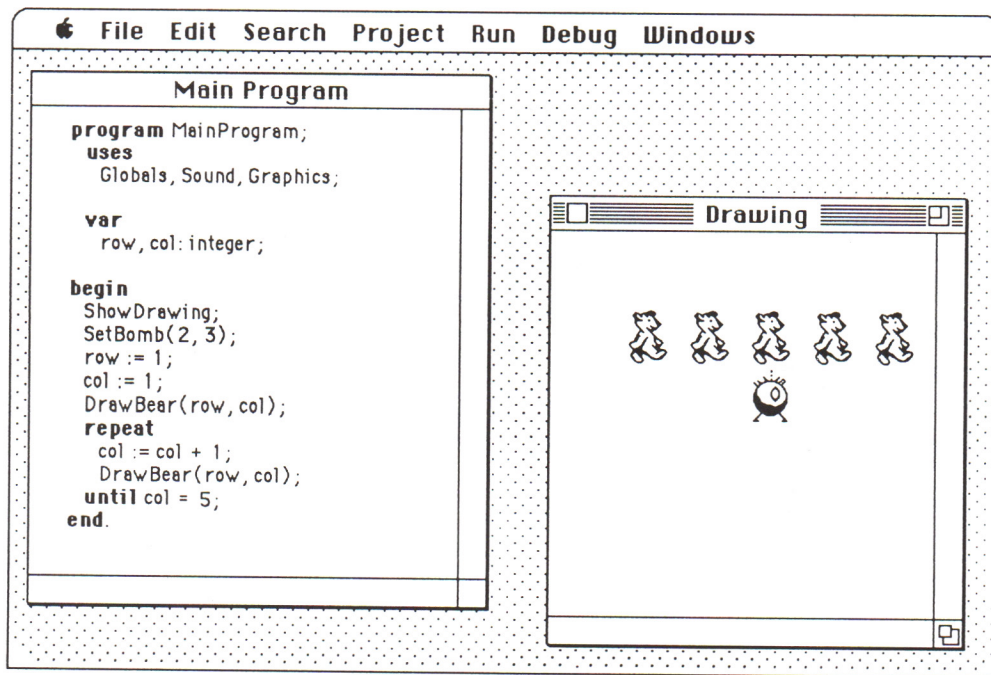
Repeat Loops

Problem

In Stage One, in order to move the bear you had to type in new values for the variables `row` and `col` and then run the program. To move the bear across the grid, you had to run the program several times. There must be a better way.

Solution: Repeat Loops

In this stage you use a programming technique that repeatedly changes the values assigned to a variable, while the program runs. This technique, called iteration, uses a **repeat** loop to draw the bear several times instead of just once.



*The main program with a **repeat** loop*

Assembly

In the assembly section of Stage Two you add a **repeat** loop to the main program.

Explanation Iteration Using Loops

Iteration is a programming technique that performs an action over and over. In this stage we accomplish this with a **repeat** loop. All **repeat** loops have the same basic structure consisting of:

- an initializing step
- the body of the loop, which includes an incrementing step
- an exit test

The initializing step sets a key variable to its starting level. Then the body of the loop is executed once. The key variable is incremented (changed in some way). A test is made on the key variable and if the test fails, the loop repeats. This continues until the test is passed and the program exits the loop.

The initializing step

The assignment statements at the beginning of the program set the starting values for the variables `row` and `col`.

```
row := 1;  
col := 1;
```

Then the statement `DrawBear(row, col)` draws the bear in the first row and first column of the grid. This establishes its initial position.

The body of the loop

The **repeat** statement is a compound statement marked by the keywords **repeat** and **until**. The two statements in between **repeat** and **until** constitute the body of the loop. These are the statements that are repeated over and over. In this loop the key variable is `col`.

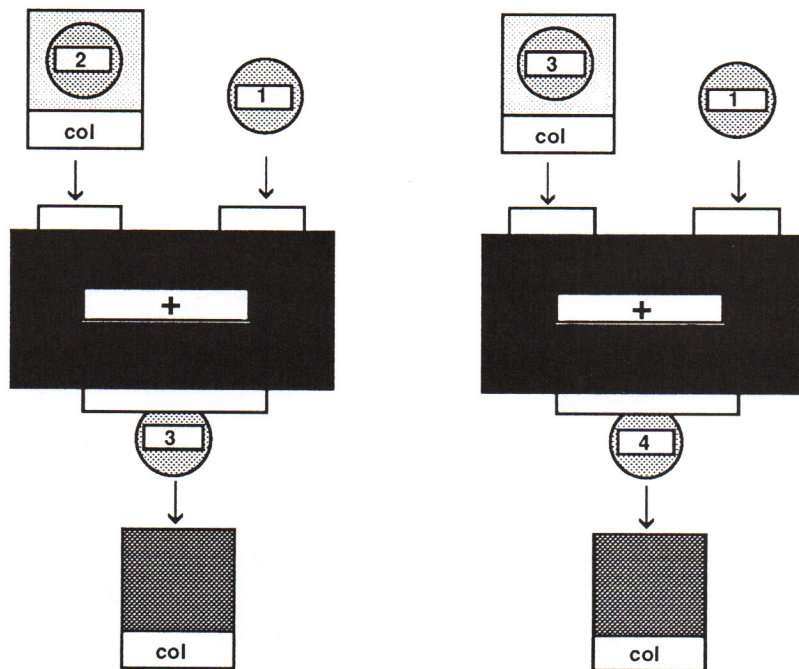
```
repeat  
    col := col + 1;  
    DrawBear(row, col);  
until col = 5;
```

The incrementing step

The first statement in the body of the loop is the incrementing step. In this case, it changes (or increments) the value of `col` by 1 each time the loop is executed.

```
col := col + 1;
```

This assignment statement consists of an **expression** on the right (`col + 1`) and a variable on the left, separated by an assignment operator. The program **evaluates** the expression on the right (it does the arithmetic) and then assigns the result of that evaluation to the variable on the left. We can think of the `+` sign as a machine that takes in two integers and produces a third.



Since this statement is inside the **repeat** loop, the integer value assigned to `col` is increased by 1 (or **incremented**) each time the program passes through the loop. As `col` changes, the bear is drawn in a different column of the grid.

The exit test

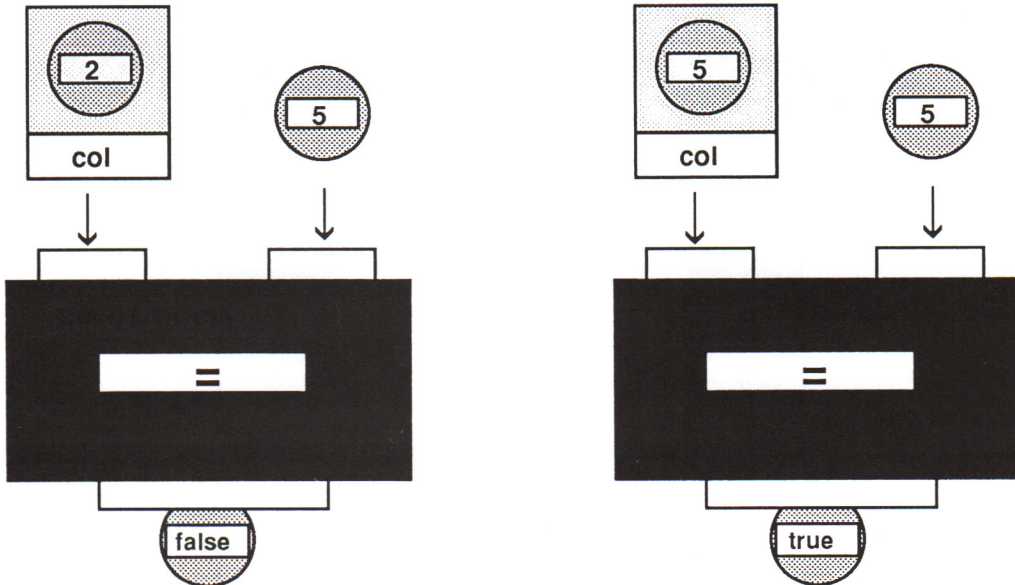
The last statement in the **repeat** loop is the exit test. It contains the statement:

```
until col = 5;
```

The body of the loop is executed repeatedly until this condition is true—until the variable `col` equals 5. At that point, the program exits the loop and continues with the statement that follows it. It is quite possible, by the way, to write a **repeat** loop that always fails the exit test. In the present example, if we started the program by assigning 6 to `col` instead of 1, the program would loop without stopping because `col` would start at the value 6 and get larger and the expression `col = 5` would never be true!

Boolean expressions

The expression `col = 5` is a **boolean** or logical expression. It evaluates as either **true** or **false**. For example, if `col` has a value of 2, the expression `col = 5` evaluates as **false** because `2 = 5` is false. When `col` has the value 5, the expression evaluates as **true**. The operator `=` is like another machine: it takes in two values and returns a third.



Flow of Control

From watching your program in action, especially when using the **Step** command as described in the Tinkering section, you probably have a sense of how statements are executed one after another. The term “flow of control” refers to the order in which statements are executed.

For the most part, control passes directly from one statement to the next until the last statement at the end of the program block (**end.**) is reached. In the **repeat** loop this process is more complicated. Here the flow cycles through the block of statements in the loop until the exit test is passed. At that point control passes out of the loop to the statement directly below the **until** statement.

Pascal Note

The words `row`, `col`, and `DrawBear` are all **identifiers**. Identifiers are names for things like constants, procedures, functions, variables, types, and so on. Pascal identifiers can be of any reasonable length, but they must start with a letter of the alphabet. Identifiers can contain numerals, letters, and the underscore character (`_`), but they may not contain spaces or most other characters. Pascal programmers often show word breaks in their identifiers by capitalizing the first letter of each word: `SetBomb` or `ShowDrawing`.

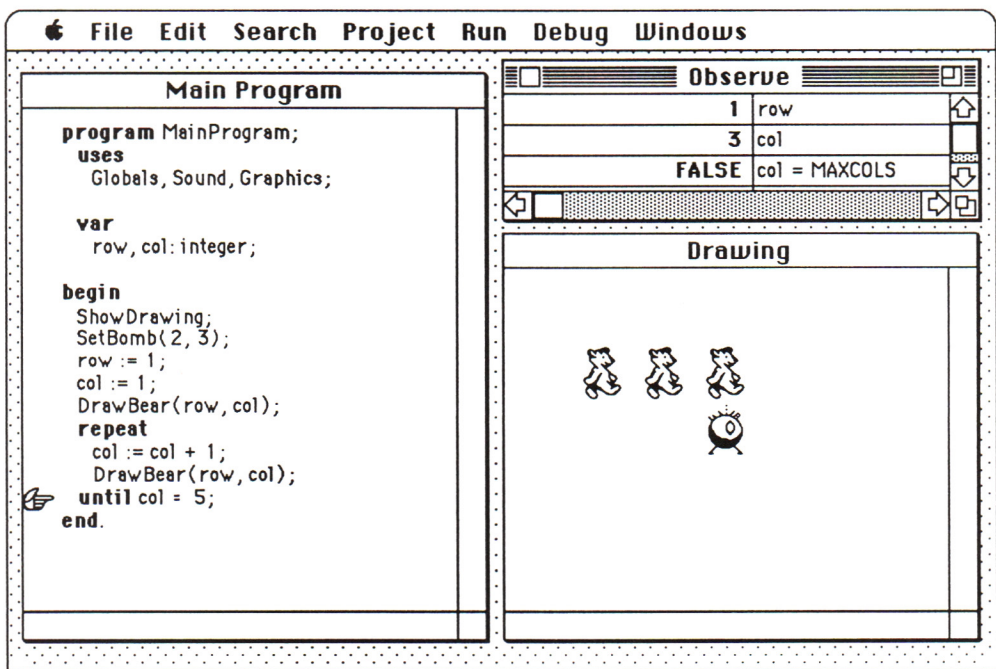
Tinkering

The Observe Window

The Tinkering section in JEP Instructions describes how to use the Observe window and **Step** command to watch the flow of control in the program. The Observe window lets you peek inside variables to see the values that are currently stored there.

Challenge: Diagonals

Can you make the bear move diagonally, from top left to bottom right? What about from top right to bottom left?



Using the Observe window

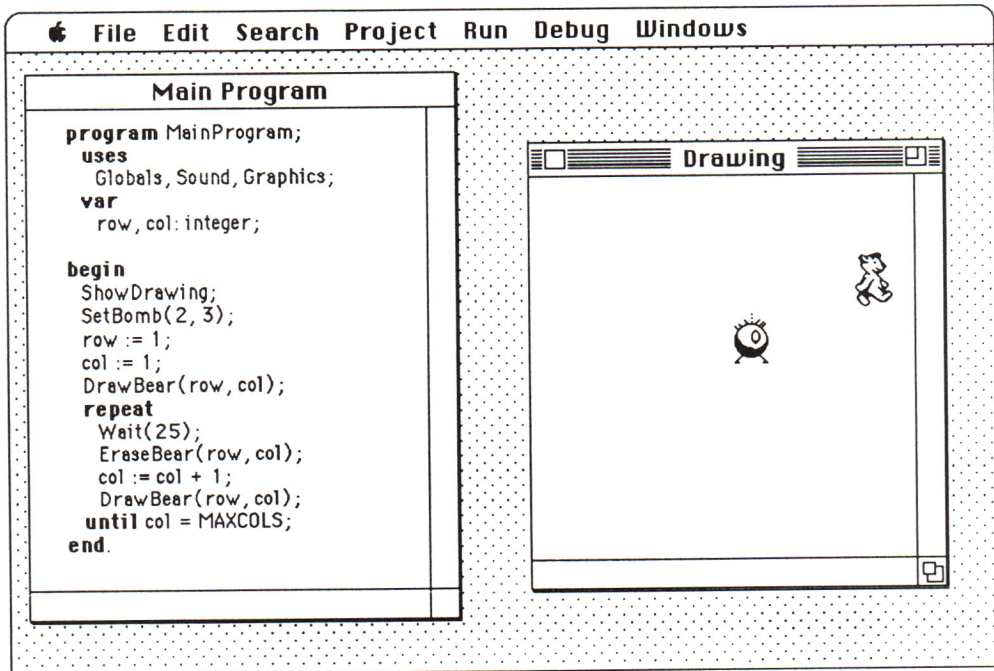
Stage Three Simple Animation

Problem

The program, with its **repeat** loop, draws the bear five times in different positions, but each image remains on the screen. Instead of a single bear *moving* across the screen, you get a row of bears. It's nice, but it's not animation.

Solution: An Erase Cycle

In this stage, you solve the problem by adding code that erases one picture of the bear before drawing the next one.



*The **repeat** loop in Stage Three*

Assembly

In Stage Three you add two statements to the **repeat** loop, one that erases part of the grid and one that simply “waits.” Both of these tasks, drawing and waiting, are performed by separate procedures. The result is a simple animation effect.

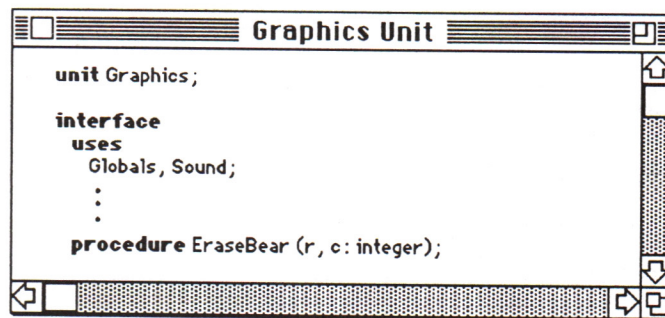
Explanation Anatomy of a Procedure

The **EraseBear** procedure erases a rectangular area in the grid. For example, `EraseBear(1, 1)` will erase a bear (or anything else) in the first row and first column of the grid.

For a closer look at `EraseBear`, open the Graphics unit. As you will see, `EraseBear` is declared in both the **interface** and the **implementation** part of the unit.

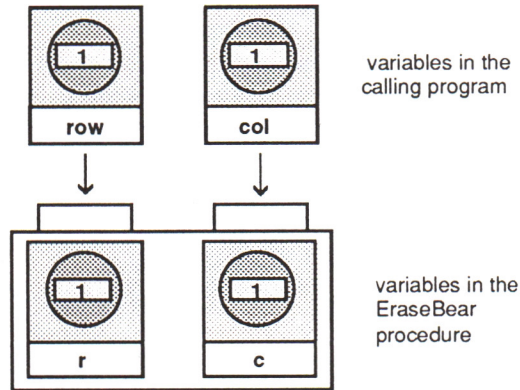
Interface

The interface part, at the top of the unit, begins with the keyword **interface**, followed by a **uses** clause. Then all the procedures defined in the unit that are called from other units are listed, along with their parameters and types.



The interface to the EraseBear procedure

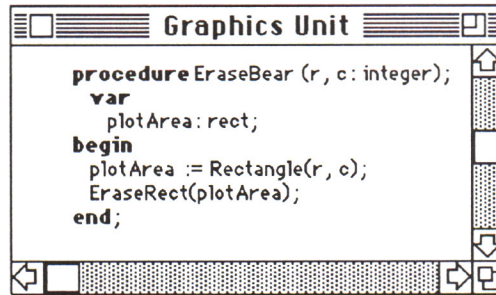
The interface tells us that the procedure `EraseBear` takes two parameters and that they must both be integers. When the main program passes these parameters to `EraseBear`, the procedure stores the values of the parameters temporarily in variables it calls `r` and `c`. The procedure uses `r` and `c` in place of `row` and `col`. Although the variable names are different, the values are the same.



The number, type and order of parameters in the procedure definition (the **formal** parameters) must match exactly the number, type and order of parameters in the procedure call (the **actual** parameters). In the procedure call to `EraseBear` in the main program, the actual parameters are the variables `row` and `col`. They match the formal parameters in the interface part of the procedure definition, defined as `r` and `c`. As a result, the value of the parameter `row` in the main program is assigned to the `r` in the procedure; the value of `col` is assigned to `c`.

Implementation

Scroll down through the unit and find the implementation part of the `EraseBear` procedure:



The EraseBear procedure

Here the procedure name is repeated, followed by the parameter list, which is optional. The procedure itself contains two statements. The first defines the area of the grid designated by the variables `r` and `c`. It stores this area in the variable `plotArea`. The second statement calls a Toolbox procedure `EraseRect` and passes it the value of `plotArea`. The cell is erased.

THINK Pascal Note

If a procedure is *not* called outside the unit in which it is defined, it need not be listed in the interface part. The parameter list is included in the first line of the procedure definition in the implementation part.

Toolbox Note

The variable `plotRect` is a `rect`, a type created by the Macintosh ToolBox for use in designating a rectangular area on the screen.

The Wait Procedure

Wait, a second JEP procedure defined in the Graphics unit, is called by this statement in the main program:

```
Wait (25);
```

When this statement is executed, control passes to the Graphics unit and the `Wait` procedure is executed—the program pauses for a specified length of time. The argument determines how long this pause will be. The unit of time in this procedure is 1/60 second, so `Wait (25)` pauses for 25/60 of a second.

Again, scroll through the Graphics unit to see the definition of this procedure. Locate its interface part and its implementation part.

Different Kinds of Procedures

`GridWalker` uses two different kinds of procedures: those that are defined by THINK Pascal in its libraries and those that are defined in the `GridWalker` project itself.

For example, the procedure `ShowDrawing` is a THINK Pascal procedure—you can use it in any program that you write because it is defined in the `Runtime.lib` library which is automatically included in every THINK Pascal project. You can also use the procedure `EraseRect`, which is one of the procedures in the Macintosh ToolBox and is defined in the library `Interface.lib`.

On the other hand, `Wait` and `DrawBear` are JEP procedures defined in the JEP Graphics unit. They can be used in the `GridWalker` program because the main program uses the Graphics unit and therefore has access to the procedures defined there.

THINK Pascal Note

For a complete list of the contents of the `Runtime.lib` and the `Interface.lib`, see the THINK Pascal *User's Manual*.

An Animation Sequence

In this stage the **repeat** loop creates a simple animation effect, which works like this:

1. Draw the bear in its initial position (1,1).
2. Begin the **repeat** loop.
3. Pause briefly.
4. Erase the bear from its current position.
5. Increment `col`.
6. Draw the bear. Since `col` has been incremented, the bear is drawn in the column to the immediate right of the previous bear.
7. Examine the value of `col`; if `col = MAXCOLS`, halt. Otherwise, repeat from step 3.

The value for `col` is not changed between `DrawBear` at the end of the **repeat** loop and `EraseBear` at the beginning of the loop. Consequently, `EraseBear` erases the same cell that `DrawBear` has just filled.

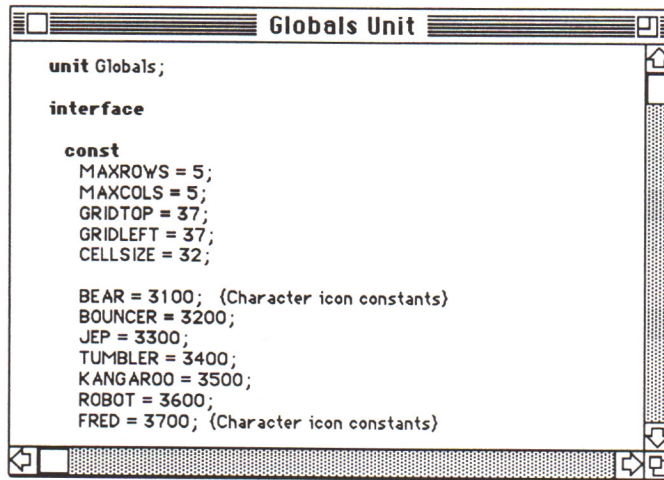
Constants

A **constant** is an identifier that is assigned a value in a constant declaration. This value cannot be changed as the program runs. In `GridWalker`, for example, the number of columns in the grid will not be changed by the program. For that reason, we use a constant to represent this number.

In the **repeat** loop, locate the statement

```
until cols = MAXCOLS;
```

`MAXCOLS` is a constant that has been declared in the `Globals` unit. Open the `Globals` unit and find the keyword **const** at the top. This marks the beginning of the constant declaration. Below the keyword you will find a list of constants that are defined for the `GridWalker` project and the values assigned to them. Because these constants are defined in the interface section, they can be used by any unit that uses the `Globals` unit.



The constant declaration in the Globals unit.

As you can see, the constant MAXCOLS has the value 5. Therefore, the **repeat** loop stops when the variable `col` equals 5.

Why use constants in a program instead of the values themselves? Because GridWalker will use the constant MAXCOLS in many different procedures, and it is much easier to change the value assigned to the constant MAXCOLS in one place (in the constant declaration) than to change every statement in which that value is used throughout the program. If you want to make the grid larger, all you need to do is open the Globals unit and change the value assigned to MAXCOLS. Try it.

About Icons

The bear, the bomb, and all the creatures in the GridWalker project are **icons** that were created for the GridWalker project. The information necessary to draw these icons is saved in a **resource file** called JEP Resources.rsrc. This file is located in the GridWalker folder and is attached to the GridWalker project.

Icons and their resource ID numbers

An icon is a small square picture, 32 pixels on a side. Each icon in the JEP resource file has a unique identification number (resource ID). For example, icon resource ID 3109 is a picture of the bear facing to our right (east) with one foot slightly forward. This particular icon is used by the DrawBear procedure.



3101



3105



3113



3109

In all, there are seventeen bear icons in the resource file, showing the bear facing in different directions, each in a slightly different pose. In Stage Six the program uses four different icons to draw the bear facing in each direction. In Stage Eleven, sixteen distinct poses are used to create a more complicated form of animation.

The `DrawBear` procedure, defined in the Graphics unit, draws the bear icon on the screen. It defines a rectangular area and then draws the picture of the bear (icon resource ID 3109) in that area. The `SetBomb` procedure, also defined in the Graphics unit, draws the icon that has the value of the constant `BOMB` (icon resource ID 8020). A list of the icon ID numbers is included in the Appendix.

Tinkering

The Resource File

The Tinkering section in the JEP Instructions describes how to attach a resource file to a project using Run Options and it demonstrates what happens when this file is not attached.

In the Tinkering section, you also study the flow of control in procedure calls using the **Step Into Calls** command.

Challenge

Can you animate the *bomb* across the screen?

Run-time Environment Settings

Resources ☒ Use resource file: **JEP Resources.rsrc**
for resources used by the project.

Text Window Text Window saves **5000** characters

☐ Echo to the printer

☐ Echo to the file:

Memory Stack size: **16** kilobytes

Zone size: **128** kilobytes

OK **Cancel**

The Run Options dialog box

Stage Four

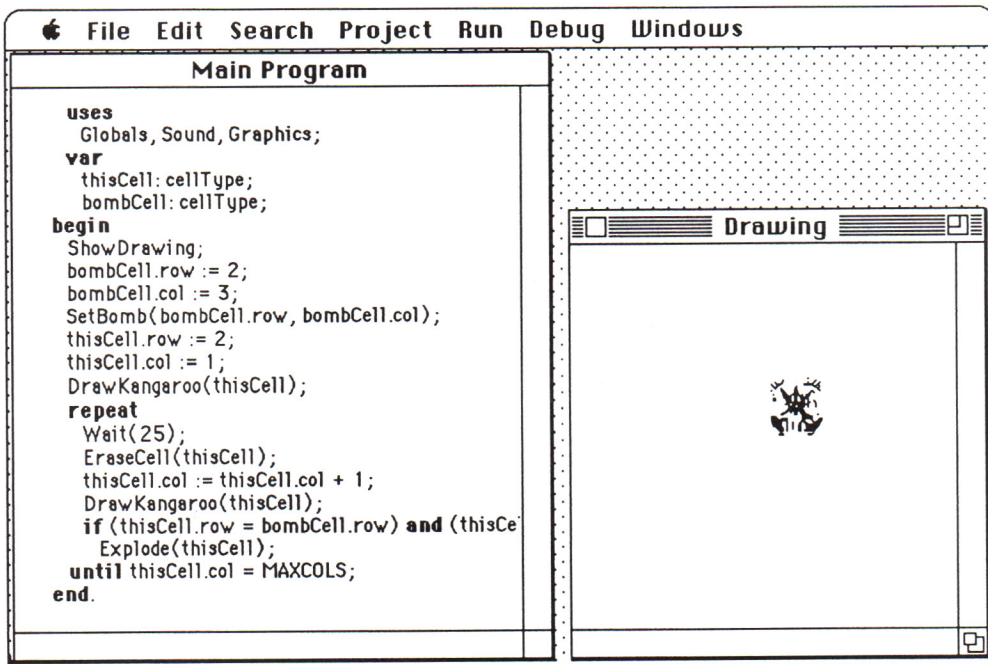
User-Defined Types

Problem

When you run the program at the end of Stage Three, the bear walks into the bomb but nothing happens. The bear was unconcerned that there was something already in that cell.

Solution: A New Type

In this stage, you add statements that determine if a cell is empty or if it contains a bomb. To help with this, you create a user-defined variable type, called `cellType`. Then you use this new variable type to refer to the intersection of a row and column in the grid as a single entity, called a "cell."



The action in Stage Four

Assembly

In Stage Four you add several components to the project, including:

- a new data type called `cellType`
- a call to the procedure `DrawKangaroo` that uses the new data type
- a different form of the **repeat** loop

Explanation Types

In Stage One, you declared two integer variables in the main program. The `integer` type is a predefined Pascal type. Other predefined types include `boolean` (which you will use in Stage Five), `real` (numbers with decimals), and `char` (letters, numerals, and so on).

Defining a New Type

Types that are not predefined can be defined in the program itself. For example, special types are defined in the library `Interface.lib` to work with commands in the Macintosh Toolbox. These include, among others, `longint`, `rect`, and `handles` (discussed briefly in Stage Seventeen).

In this stage, you declare a new type, called `cellType`, designed especially for the `GridWalker` program. This new type organizes information with a useful Pascal structure called a **record**. Later, you will create types using two other Pascal structures, **enumerated lists** (Stage Six) and **arrays** (Stage Nine).

The record type

The type `cellType` is a user-defined type. It might also be called a “programmer-defined” type. To see the definition, open the `Globals` unit and find the **type** declaration, just below the constant declaration. Locate these lines:

```
type
  cellType = record;
    row: integer;
    col: integer;
  end; {cellType declaration}
```

This part of the unit declares (defines) new types that the program can use. It begins with the keyword **type**. For the time being, only one type is declared here. The next line begins the declaration of the type `cellType` and identifies it as a **record**. The next two lines identify the two **fields** of the record and define the type of data that each field can contain. The last line designates the end of this particular type declaration.

A **record** type is composed of a number of fields, and each field is itself of a specified type. For example, both fields in `cellType` are integers. You can think of a `cellType` variable as a box with two compartments.

In this record the two integer fields are called `row` and `col`. They correspond to the row and column position of a cell in the grid. The fields in this variable are used in the way that the integers `row` and `col` were used in previous stages.

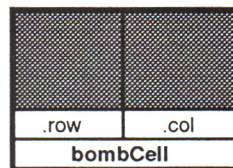
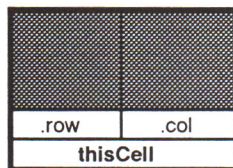
Declaring variables

Once you have defined a new type, you can declare variables of that type. Here the new type appears in the variable declaration part of the main program:

```
var
  thisCell: cellType;
  bombCell: cellType;
```

A single line can also declare two variables of the same type:

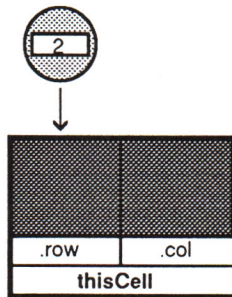
```
thisCell, bombCell: cellType;
```



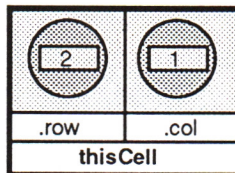
Assigning Values to a Record Variable

The variable `thisCell` has two fields, `row` and `col`. The first field is represented by the expression `thisCell.row`. The value 2 is assigned to the `row` field of the record `thisCell` with the statement

```
thisCell.row := 2;
```



A second statement assigns the value 1 to the `col` field. Other statements assign values to the record variable `bombCell`. After both assignments are made and we look inside the variable, we can see the values stored in the two fields.



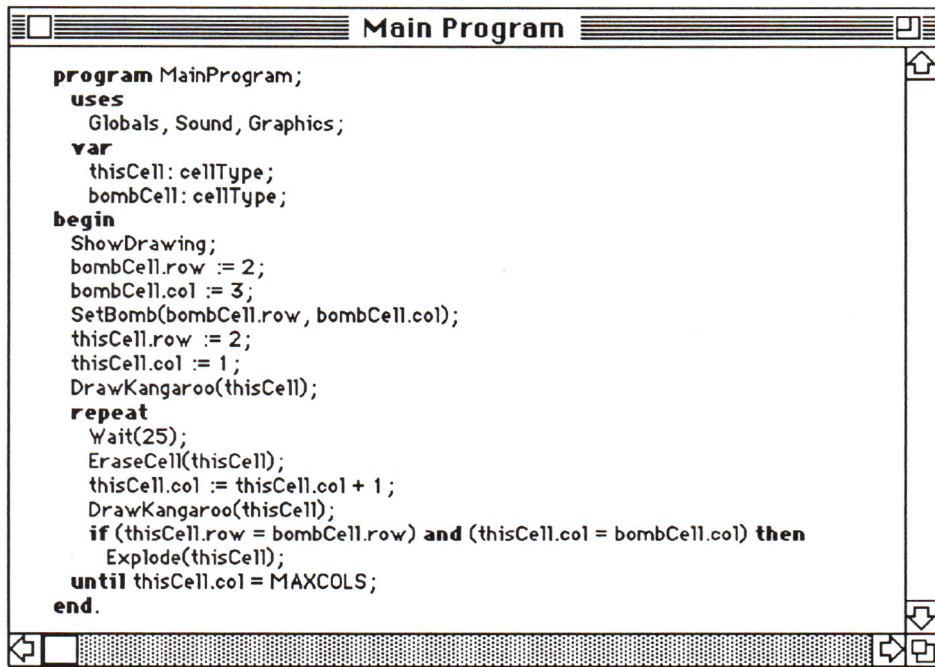
In the revised **repeat** loop in the main program, the character's column position is changed by changing the value stored in the `col` field of the record `thisCell`:

```
thisCell.col := thisCell.col + 1;
```

Also, the **until** statement uses `thisCell` as well:

```
until thisCell.col = MAXCOLS;
```

Examine the main program to see all the components of this **repeat** loop.

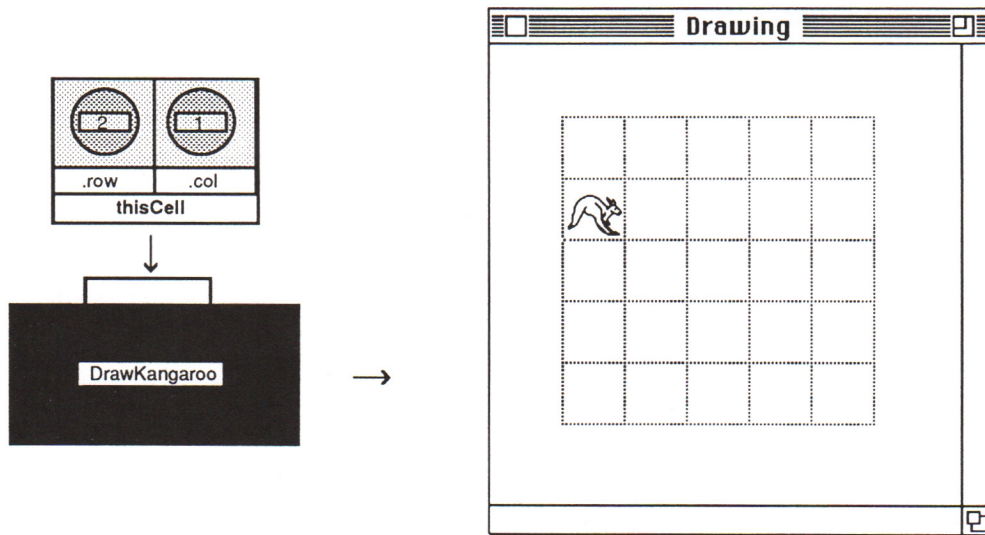


The complete main program

Passing Information in a Record to a Procedure

The usefulness of the new type becomes clear when the `cellType` variables are used in a procedure call. For example, the following statement in the main program passes all the necessary information about a cell to the procedure `DrawKangaroo` using a single parameter:

```
DrawKangaroo(thisCell);
```

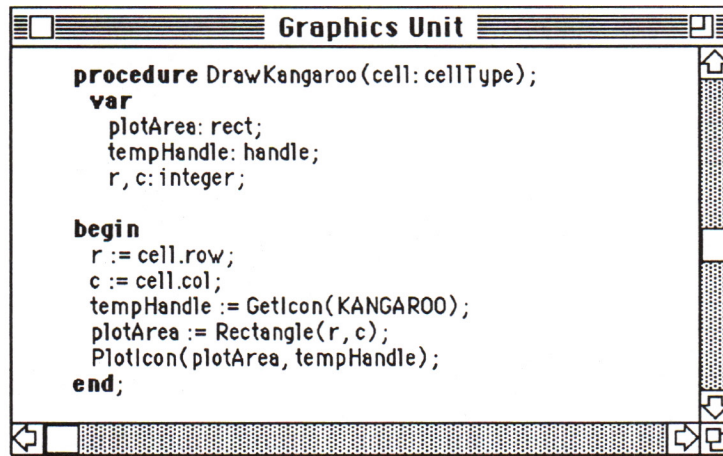



To see how a `cellType` variable is used by `DrawKangaroo`, open the Graphics unit and locate the interface part at the top of the unit:

```
procedure DrawKangaroo (cell: cellType);
```

This statement declares the procedure `DrawKangaroo` and says that `DrawKangaroo` expects to be passed a `cellType` value, which it will store in the local variable `cell`.

Scroll down through the Graphics unit, into the implementation part, to where the operation of `DrawKangaroo` is defined. This procedure is very similar to `DrawBear`. The main difference is that information about the position of the icon is passed to `DrawKangaroo` in a single parameter (the record), whereas `DrawBear` requires two separate parameters, one for the row position and one for the column position.



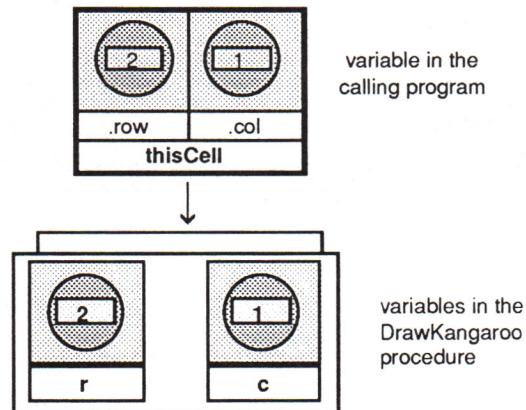
Implementation of DrawKangaroo

DrawKangaroo assigns each field in cell to a local variable in the procedure with the lines

```

r := cell.row;
c := cell.col;

```



Here *r* and *c* are local variables used by DrawKangaroo to record the row and column of the cell. Since the constant KANGAROO is 3900, the procedure then draws the kangaroo icon (icon resource ID 3900) in the corresponding cell in the grid.

Pascal Note

The DrawKangaroo procedure defines the variables plotArea, tempHandle, r and c. These variables are called **local** variables because they can only be used by DrawKangaroo. They cannot be used when the procedure finishes.

For Loops

Explode is a JEP procedure, defined in the Graphics unit. It takes a single cellType argument and produces the “explosion” animation in the specified cell of the grid. To create this animation, it uses a different iterative technique, the **for** loop, to draw three icons (ID 8021-8023) in the specified cell, pausing briefly with each iteration.

Here is how the **for** loop works. To show an exploding bomb the procedure must draw icon #8021, pause, erase the icon, draw the next icon, pause, erase, and so on. We could accomplish this by writing separate statements for each step. Or we can use a **for** loop that counts to 3:

```
for i := 1 to 3 do
  begin
    icn := BOMB + i;
    tempHandle := GetIcon(icn);
    PlotIcon(plotArea, tempHandle);
    Wait(15);
    EraseRect(plotArea);
  end; {...for loop}
```

The keyword **for** starts the loop. First it assigns the value 1 to the counter variable *i*. Then it starts to execute the body of the loop. The integer *icn* is assigned the value 8021 (the constant BOMB is 8020). The corresponding icon is plotted (the PlotIcon procedure), the program pauses, and the plotted icon is erased.



Then the counter variable *i* is **incremented**—it is assigned the value 2—and the body of the loop is executed again. Next, *i* is assigned the value 3 and the loop executed a third time. Finally, *i* is assigned the value 4, and since it is no longer in the range 1-3, the loop is finished. Control passes down to the statement following the loop block.

If Statements

An **if** statement in Pascal contains four parts:

- the keyword **if**
- the condition—a Boolean expression
- the keyword **then**
- the conditional code

The condition

If the conditional expression evaluates as **true**, the conditional code is executed; if it evaluates as **false**, the conditional code is ignored. To illustrate, look carefully at the **if** statement in the main program:

```
if (thisCell.row = bombCell.row) and
    (thisCell.col = bombCell.col) then
    Explode(thisCell);
```

The logical condition in this statement is easy to understand. It returns **true** only when both the row and the column positions of the bomb and the character are the same. If the character and the bomb are in different cells (if either the two rows or the two columns are different), the condition is not met.

Pascal Notes

Operators: The keyword **and** in the boolean expression above is a logical operator that connects two boolean expressions. The resulting conjunction evaluates as **true** only if both expressions are **true**; if either is **false**, the conjunction is **false**.

Other logical operators include **or** (which evaluates as **true** when *either* expression is **true**) and **not** (which negates the expression that follows it).

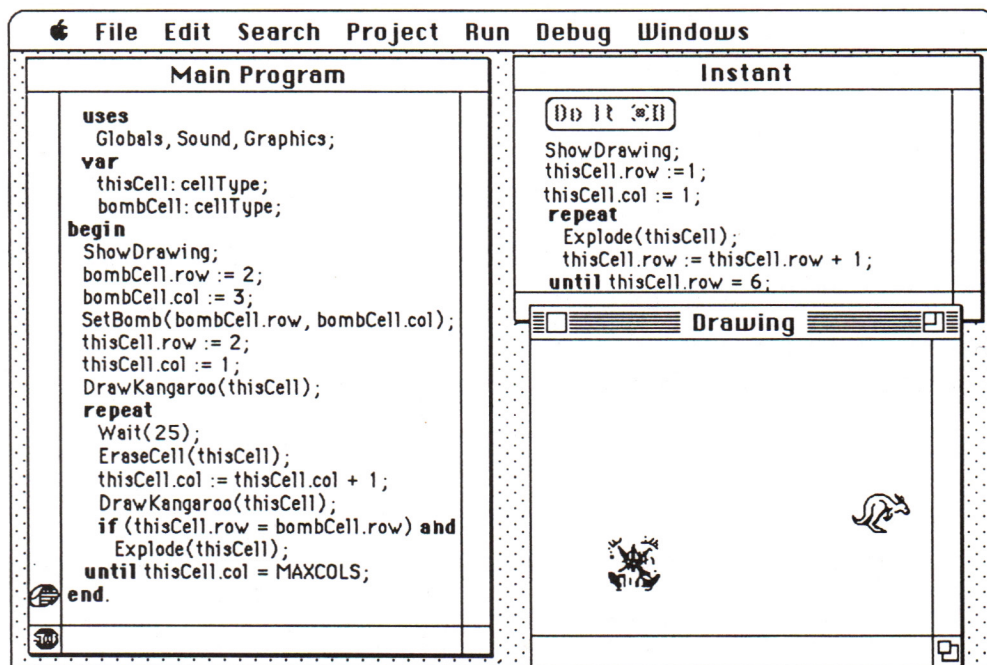
Precedence: Notice that the conditional expression above used two operators, “=” and “**and**.” Rules of **precedence** define the order in which operators are evaluated. Three rules must be followed:

1. Expressions within parentheses are evaluated first.
2. When an expression contains two operators, the one with the higher precedence is evaluated first. The unary operators (**@** and **not**) have the highest precedence; next are the “multiplying” operators (*****, **/**, **div**, **mod**, **and**); followed by the “adding” operators (**+**, **-**, **or**); and lowest are the relational operators (**=**, **<>**, **<**, **>**, **<=**, **>=**, **in**).

3. When an expression is written between two operators of the same precedence, the expression is bound to the operator on the left.

Tinkering Instant Window

The Instant window lets you execute additional program statements without compiling the main program again. The Tinkering section gives step-by-step instructions for using the Instant window.



The Instant window in action

Challenge

Can you make the kangaroo hop *over* the bomb?

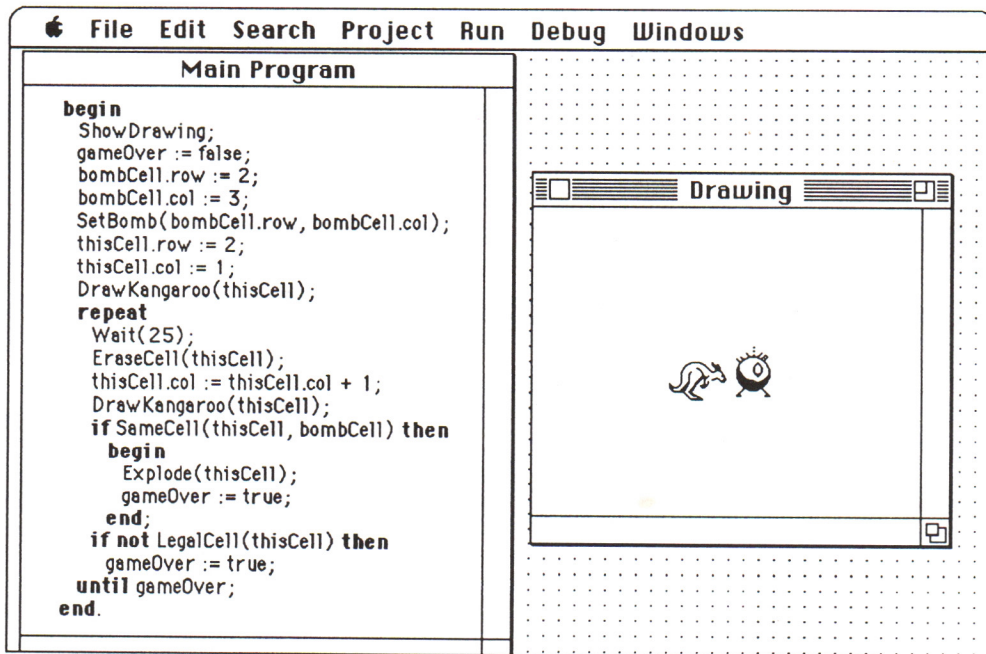
Stage Five Functions

Problem

The program can now detect those situations when the kangaroo and the bomb occupy the same cell. The problem is that the kangaroo just keeps hopping along after the explosion, as if nothing had happened.

Solution: The Function LegalCell

In this stage you add logic to the program that stops that kangaroo dead in its tracks. This logic uses a function called `LegalCell`.



The main program showing the use of LegalCell

Assembly

In Stage Five you add:

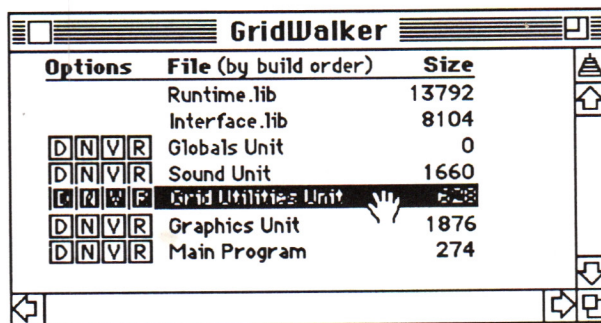
- a new unit to the project called Grid Utilities
- an `if` statement that calls the function `LegalCell`

Explanation Changing the Build Order

In this stage, you add the Grid Utilities unit to the project. It contains several new functions and procedures:

- `LegalCell`
- `SameCell`
- `InGrid`
- `NextCell`
- `RightTurn`

Since the Grid Utilities unit uses constants, types, and variables defined in the Globals unit, it must be *below* Globals in the build order. And because the functions and procedures in Grid Utilities are used by the main program, Grid Utilities must be *above* the main program. When you add a new unit to the project, it is placed at the bottom of the list. You position it in the correct build order by dragging the unit name in the Project window above the names of the other units that need to interface with it.



When you add a unit to the project, click inside the "V" and "R" boxes to enable these debugging options. The "V" will watch for overflows in integer arithmetic and the "R" checks the ranges of indexed variables.

Uses clause

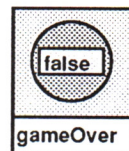
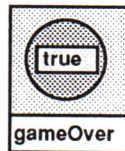
As you know, the **uses** clause at the top of a unit establishes which units in the project can be used by that unit. For example, the **uses** clause in the main program now names four units:

```
uses
    Globals, Sound, Graphics, GridUtilities;
```

The units named in the **uses** clause must all be above that unit (or main program) in the build order.

Boolean Variables

The variable `gameOver` is a variable of the type **boolean**. It can be assigned one of two values, `true` or `false`. This variable is used in the main program to stop the **repeat** loop once the kangaroo hits the bomb. It is initially assigned the value `false`, but the value changes to `true` if the kangaroo goes out of the grid or runs into the bomb. When `gameOver` is `true`, control passes out of the **repeat** loop and the program ends.



Global Variables

Because the variable `gameOver` is defined in the interface part of the `Globals` unit, it can be used by any unit that comes after `Globals` in the build order and includes `Globals` in its **uses** clause. We have purposely put the `Globals` unit first in the build order and listed it in the **uses** clause of every unit. Therefore, all variables defined in the interface part of the `Globals` unit can be used in all other units, including the main program.

The variables declared in the main program, in contrast, cannot be used in other units. They are said to be **local** to the main program. More generally, any variable is **local** to both the unit that it is defined in and to the units that use it. The variables declared in the `Globals` unit, on the other hand, are said to be **global** to the entire project.

Constants and types are also either local or global. Constants and types can be used by the unit in which they are declared and by any unit that uses that unit.

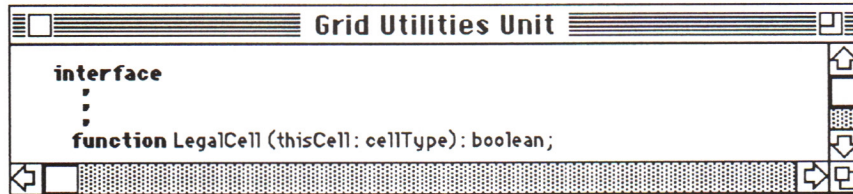
Functions

A function is a particular kind of procedure. Like a procedure, a function includes a set of statements that perform a specific task.

Functions differ from procedures in that they **return** a single value which can then be used by other parts of the program. This value is assigned to the function name.

Examine the function `LegalCell` in the Grid Utilities unit.

Interface part



The interface to the LegalCell function

The expression in parentheses indicates that the function `LegalCell` takes one parameter of the type `cellType` and stores that value in a variable it calls `thisCell`. The expression following the colon indicates that the function will return a boolean value (`true` or `false`).

In other words, you can pass `LegalCell` a cell, and it will pass back a boolean value that tells you if that cell is in the grid (`true`) or outside the grid (`false`).

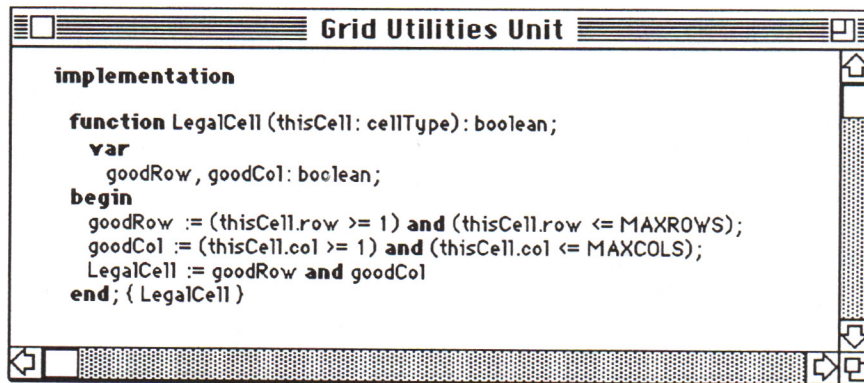
Implementation

Examine the implementation part of the Grid Utilities unit by scrolling down. Here you'll find the full definition of the function.

The function first declares two boolean variables, `goodRow` and `goodCol`. These variables are local to the `LegalCell` function. Then the function assigns the variable `goodRow` a boolean value in the following statement:

```
goodRow := (thisCell.row >= 1) and (thisCell.row <= MAXROWS);
```

The logical expression in this statement is true only when both expressions in parentheses are true. Therefore, `goodRow` is assigned `true` only when the `row` field of `thisCell` is in the range 1 to `MAXROWS`. Otherwise it is assigned the value `false`. A similar assignment is made to the variable `goodCol`.



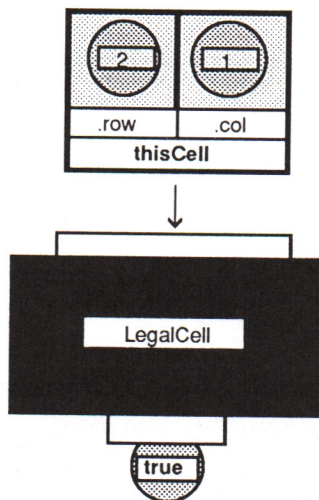
The implementation of LegalCell

The result of the function

In any function, the statement that assigns a value to the function name determines the result returned by that function. The statement at the end of the implementation of LegalCell assigns a boolean value to the function name:

```
LegalCell := goodRow and goodCol;
```

This value is `true` when both `goodRow` and `goodCol` are `true`, and `false` in all other cases. The value assigned to `LegalCell` in this statement is the result that is returned to the calling statement.



Using Functions

Because a function returns a value, it can be used as an expression in a statement. When the program reaches a statement containing a function, the function is called, its value determined, and the statement is executed accordingly.

For example, consider this statement in the main program:

```
if not LegalCell(thisCell) then
    gameOver := true;
```

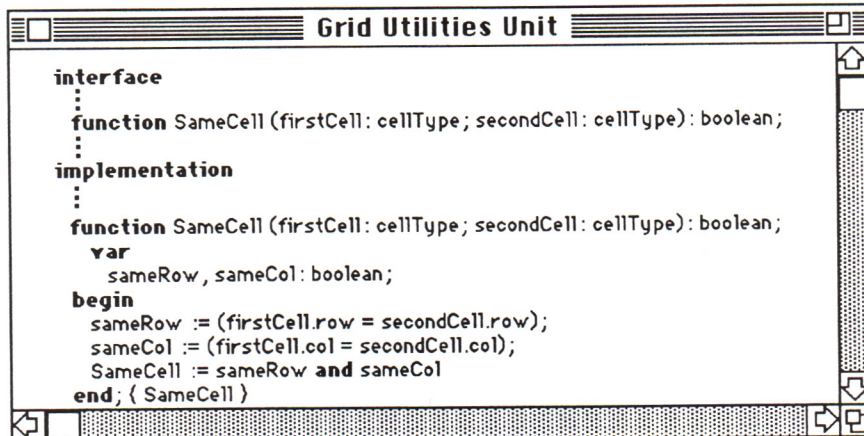
First, the current location of the icon—the information in the record `thisCell`—is passed to the `LegalCell` function. `LegalCell` returns `true` if the cell is within the boundaries of the grid and `false` if it isn't.

Next the entire conditional expression `not LegalCell(thisCell)` is evaluated. When `LegalCell(thisCell)` returns `true`, `not LegalCell(thisCell)` is `false`.

Finally, the `if` statement is executed. When the test condition is `true`, `gameOver` is assigned the value `true`. When it is `false`, `gameOver` does not change.

Another Function

The JEP function `SameCell`, defined in the Grid Utilities unit, takes two cells as arguments.



The function SameCell

As you can see, `SameCell` returns `true` if both the `row` fields and the `col` fields of the two cells are the same. It returns `false` if either condition is unmet. Compare these two expressions:

```
if (bombCell.row = thisCell.row) and  
    (bombCell.col = thisCell.col) then...
```

```
if SameCell(bombCell, thisCell) then...
```

Both expressions accomplish the same thing. The second is easier to read because the logic of comparing two cells is tucked away in the function SameCell.

Tinkering Trace and View Options

The Tinkering section describes how to use the **Trace** command and the **View Options** command on the Run menu.

Challenge

When the constant MAXCOLS in the Globals unit is 5, the kangaroo stops in the 6th column. Why? Can you get it to stop in column 5?

Can you get the kangaroo to *reverse direction* when it comes to the edge of the grid?

Stage Six

“Character” Records

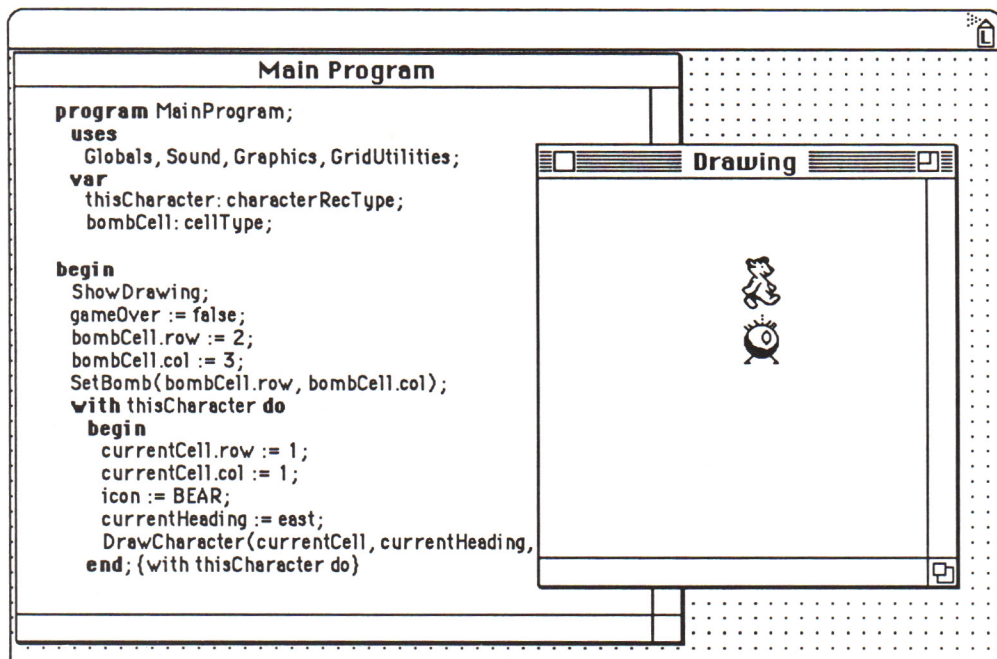
Problem

The program now keeps track of a single piece of information about a moving icon: its current position in the grid. However, as the program grows more complex, it must keep track of other facts as well. In addition to the current cell position, it must record:

- the current heading—the direction the icon will move next
- the icon—which icon it is
- the strategy—how it moves through the grid

Solution: Defining Another Type

It would be nice to store all of this information in one place. To do that you create a new data type called `characterRecType`. This type defines fields for each piece of information about what we will call a “character.”



Assembly

In Stage Six you add:

- a data type with a record structure called `characterRecType`
- a statement that assigns a value to the new type
- a change to the **repeat** loop that takes advantage of the new data type

Explanation Creating a New Type

Creating a new variable type is a two-step process. First, you declare the new type, `characterRecType`. Then you declare a variable, `thisCharacter`, of that type.

`characterRecType`

The new data type, declared in the `Globals` unit, is called `characterRecType`:

```
characterRecType = record
    currentCell: cellType;
    currentHeading: headingType;
    icon: integer;
end; {characterRecType declaration}
```

The first line of this definition names the type and identifies it as a **record** type. The next three lines identify the three **fields** of the record and define the type of data that each field can contain. The last line designates the end of this type declaration.

.row	.col	.current- Heading	.current- Icon
.currentCell			
thisCharacter			

Variable declaration

The variable declaration part of the main program can now declare a single variable `thisCharacter` of the new type.

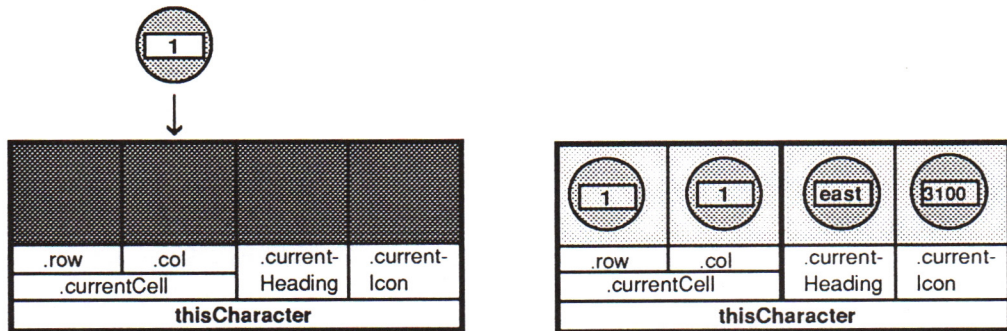
```
thisCharacter: characterRecType;
```

The information previously stored in `cell` can now be stored in the `currentCell` field of the variable `thisCharacter`. We can also store information about a character's direction of movement in the `heading` field and its icon resource ID in the `icon` field.

Assigning values to the record

The fields in the record are assigned values. For example, this statement assigns the value 1 to the `col` field of the record `thisCharacter`:

```
thisCharacter.currentCell.col := 1;
```



Notice that the first field of the `characterRecType` is itself a record (`cellType`) that contains two fields. As with other variables, we can examine the contents of the fields in this record after the assignments are made using the Observe window (Tinkering Stage 3) or LightsBug (Tinkering Stage 9).

Enumerated Types

The heading field makes use of another user-defined type, called `headingType`, which is also declared in the Globals unit. It makes use of another special Pascal structure called an **enumerated type**. In declaring this type, all possible values for variables of this type are listed in parentheses after the name of the type.

Open the Globals unit and examine this line in the type declaration part:

```
headingType = (west, north, east, south, noHeading);
```

Since there are only five different values for headings, this is an efficient way of managing variables of this type. Just as a boolean variable has two possible values, `true` and `false`, a `headingType` variable has five possible values: `west`, `north`, `east`, `south`, and `noHeading`.

More Constants

Constants for the different character icons are declared in the `Globals` unit. Find these lines in the constant declaration part:

```
BEAR = 3100;  
BOUNCER = 3200;  
JEP = 3300;  
TUMBLER = 3400;  
KANGAROO = 3500;  
ROBOT = 3600;  
FRED = 3700; {Character icon constants}
```

As explained earlier, these constants refer to the resource IDs for icons stored in the resource file, `JEP Resources.rsrc`, attached to the `GridWalker` project. With these constants defined in the `Globals` unit, we can refer to the icons by name rather than by number.

Pascal Note

Following a well-established Pascal convention, we use uppercase letters for all constant identifiers (`BEAR`). Procedure and function names begin with an uppercase letter (`DrawBear`). Names of variables and variable types begin with lowercase letters (`thisCharacter`). Throughout, uppercase letters denote word breaks when an identifier contains more than one word. These conventions are used only to make the code more easily read—the compiler ignores case entirely.

The With Structure

To assign values to the record `thisCharacter`, we named both the record and the field in the assignment statements. Since there are now several statements that name the same record, this gets repetitious:

```
thisCharacter.currentCell.row := 1;  
thisCharacter.currentCell.col := 1;  
thisCharacter.icon := BEAR;  
thisCharacter.currentHeading := east;
```

Pascal provides a shortcut, the **with** statement to deal with this repetition. In a block that begins with the keyword **with**, all variables within the block are automatically interpreted as fields in a record.

The following block uses the **with** statement with the record `thisCharacter`.

```

with thisCharacter do
  begin
    currentCell.row := 1;
    currentCell.col := 1;
    icon := BEAR;
    currentHeading := east;
    DrawCharacter(currentCell, currentHeading, icon);
  end;

```

In this example, all variables are first considered as fields in the record `thisCharacter`. For example, the `icon` field of the record is set to the constant `BEAR` (icon resource ID 3100), the `row` and `col` fields are set to 1, and the `heading` field to `east`. The last line in the block draws the bear in the specified cell, facing east. Any variables in the block that are not fields in the `thisCharacter` record are evaluated as regular variables.

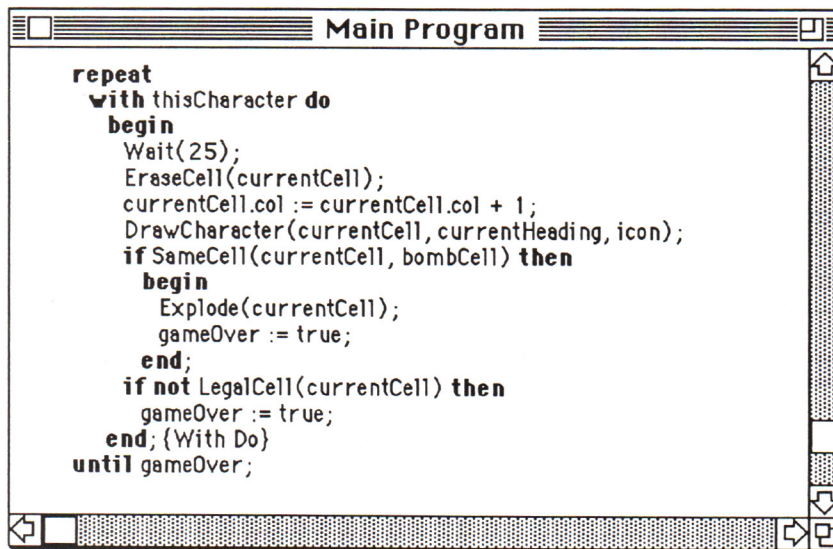
The **repeat** loop in the main program also uses the **with** statement with the record `thisCharacter`.

The Revised Repeat Loop

Up to this point, you drew icons with the specialized procedures `DrawBear` and `DrawKangaroo`. In this stage, you use a more general procedure, `DrawCharacter`, to draw any character in any heading. It takes three parameters, a cell, a heading, and an integer assumed to be an icon ID.

The **repeat** loop contains a second **with do** block. The `row` and `col` fields in `thisCharacter` are used to move the character around the grid. In all other respects the **repeat** loop is the same as before.

In Tinkering, you use this more general procedure to explore the four headings for different character icons.



*The repeat loop showing the **with do** structure*

Pascal Note

You may have noticed words in braces { } in the source code throughout GridWalker. These words are **comments**. They are added to the source code to make it easier to understand, but they are ignored by the compiler. In the figure above, the end of the **with** block is marked with a comment. When you use comments, always begin with a left brace and end with a right brace. Each comment must fit on a single line.

Tinkering Different Commands

The Tinkering section introduces the **Source** options command, the **Reset** command and the **Go-Go** command.

Challenge

Try to get the character to move in the direction it is heading. For example, if it is heading north, have it move up one row for each pass through the **repeat** loop.

Stage Seven

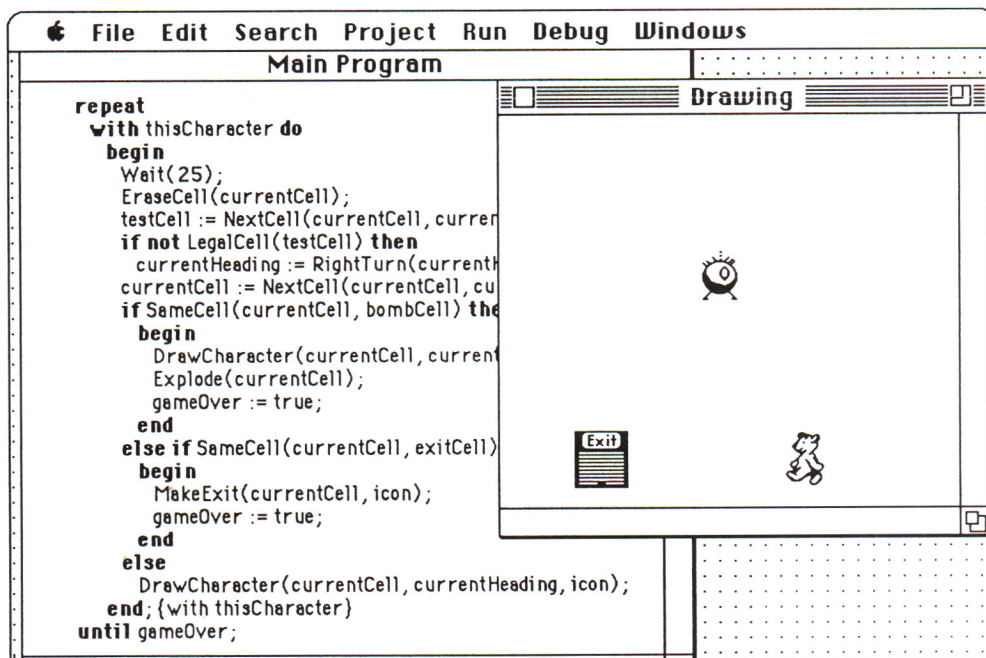
Simple Algorithms

Problem

Up to this point the character can move along only one path—along a single row or up and down a single column. Negotiating a maze filled with obstacles will demand more sophisticated maneuvers.

Solution: A Simple Algorithm for Turning

In this stage you add procedures that a character can use to skirt the edge of the grid and turn to the right when necessary.



The main program at Stage Seven

Assembly

In this assembly stage, you add:

- two new variables of type `cellType`
- the `SetObstacle` procedure
- additions to the **repeat** loop

Explanation Algorithms

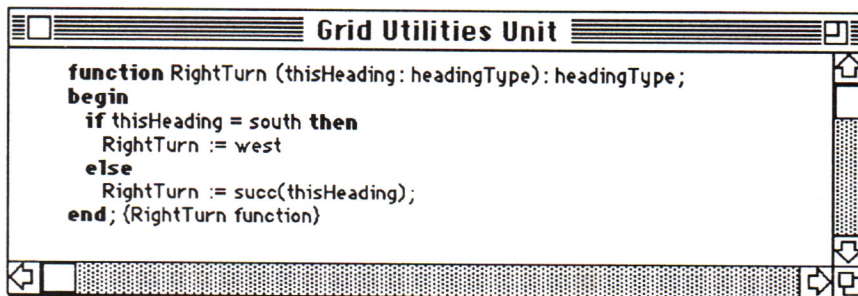
An **algorithm** is a step-by-step method for accomplishing a particular task. In this stage you add statements to the program that define a very simple (and imperfect) algorithm for finding the exit in a maze. This algorithm can be paraphrased, "Go straight ahead if you can; if you can't, turn right."

The main program implements this algorithm in the **repeat** loop. Before examining the algorithm, we must look at two new functions.

A Function Using `succ`

The main program now makes use of the function `RightTurn`. To see how `RightTurn` works, open the Grid Utilities unit.

`RightTurn` takes an integer as an argument and stores it in a local variable called `thisHeading`. It returns a `headingType` variable as a result and stores it in `RightTurn`. In other words, you can give this function a heading (like north), and it will give you back the heading that results from a right turn (east).



The RightTurn function

If the heading passed to `RightTurn` is south, then `RightTurn` returns west. Otherwise, `RightTurn` uses the `succ` function to determine the heading that results from turning right.

The function `succ`

The function `succ` is defined in the library `Runtime.lib` along with other standard Pascal functions and procedures. When given an entry in a list, `succ` returns the successor entry. Given the order of headings in the list

```
headingType = (west, north, east, south, noHeading);
```

the successor of each entry is the “right turn” from that entry. The successor of west is north, the successor of north is east, and so on. The heading south is treated as a special case—the logic of the **if then else** statement used here is described in detail in Stage Nine.

You cannot take the successor of the last entry to a list. In `GridWalker`, `noHeading` is always treated as a special case.

Note: The `pred` function is similar to `succ` except that it returns the predecessor entry in the list.

The Case Structure

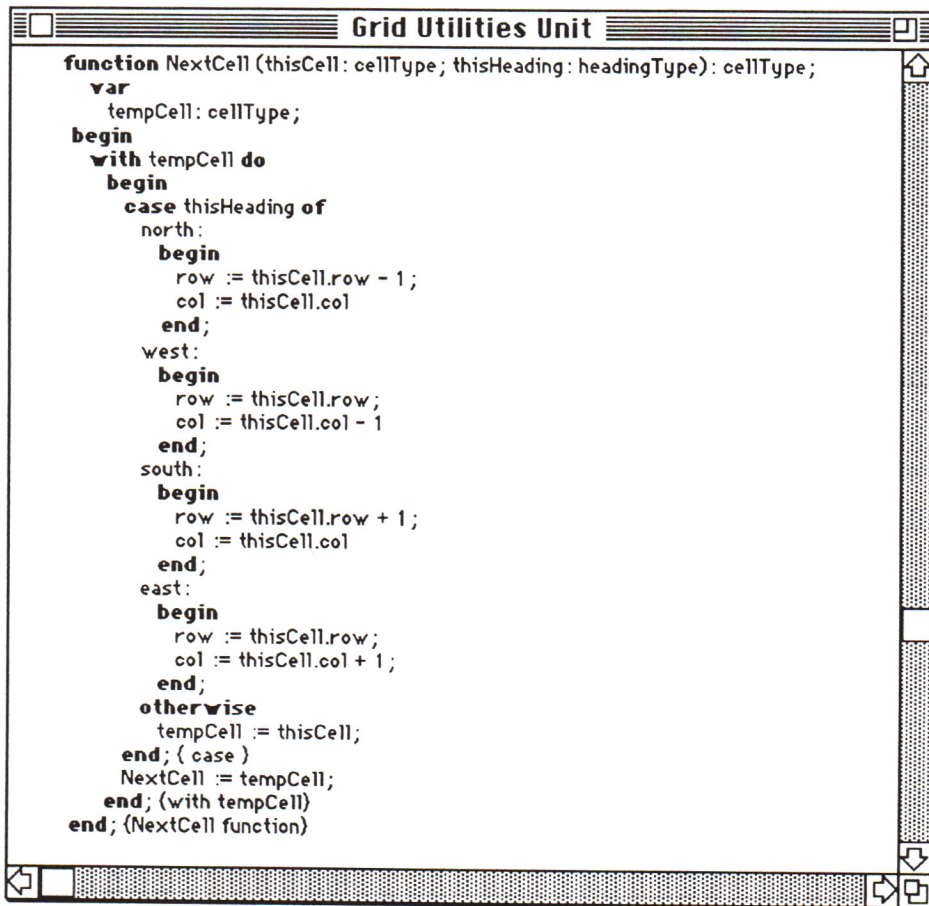
The `NextCell` function in the `Grid Utilities` unit makes use of a control structure called a case statement. This function takes two arguments, a cell and a heading, and returns the “next” (adjacent) cell in that heading.

The **case** statement begins with the keyword **case**, followed by a **selector** variable (`thisHeading`), followed by the keyword **of**. This is followed by a list of values (north, west, south, east). Each value is in turn followed by program statements.

First, the function evaluates the selector variable. Then it locates the value in the list that matches the selector variable and executes the statements that follow that value. If no value matches, the statements following the keyword **otherwise** are executed. To prevent errors, a **case** statement must either list all possible values for the selector variable or it must include the **otherwise** case.

To illustrate, suppose that the value of the selector variable `thisHeading` is west. In that case, the statements that follow the west case (and only those statements) are executed.

```
west:
  begin
    NextCell.row := thisCell.row;
    NextCell.col := thisCell.col - 1
  end;
```



A portrait of the function NextCell

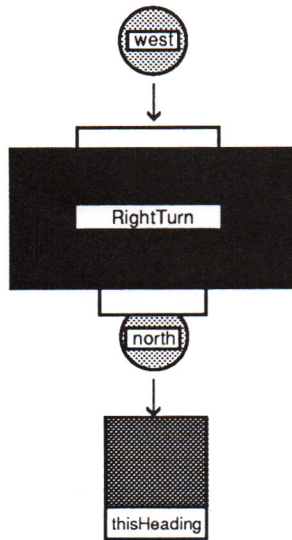
When the character is heading west, the next cell is in the same row as the current cell and in the column immediately to the left (`thisCell.col - 1`). These statements assign the appropriate values to the fields of `NextCell`. Other cases are handled similarly.

Pascal Note

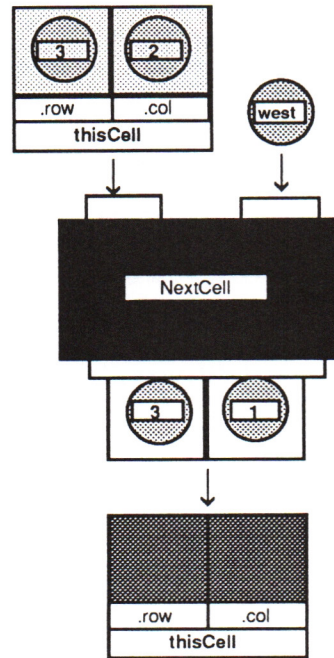
If a value in the **case** statement is followed by more than one statement, these statements must be formatted as a single block with **begin** and **end** statements. If the value is followed by a single statement, then the **begin** and **end** statements are unnecessary.

Using these functions

The `RightTurn` and `NextCell` functions are used together in the algorithm described below. Both functions take arguments and return a value. That value is assigned to a variable.



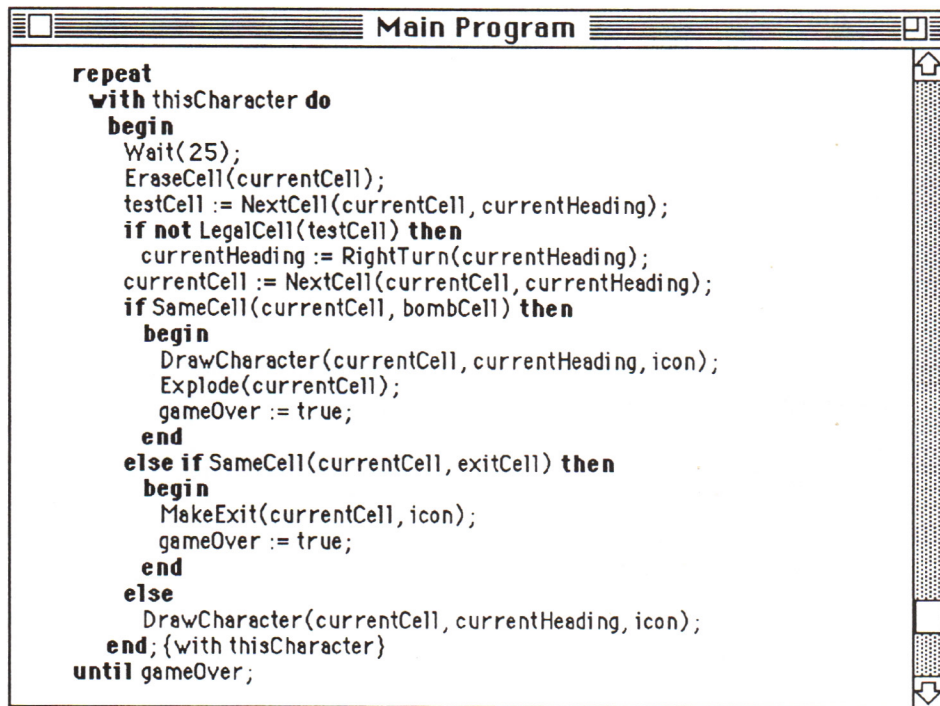
```
thisHeading := RightTurn(west);
```



```
testCell := NextCell(thisCell,west);
```

The Turn Right Algorithm

The main program uses `NextCell` and `RightTurn` to create a simple algorithm for finding the exit in a maze.



*The revised **repeat** loop*

Each time through the loop, a simple turning strategy is invoked: First, a value representing the cell in front of the character is placed in a local variable called testCell:

```
testCell := NextCell(currentCell, currentHeading);
```

Next, testCell is tested to see if it's in the grid; if it is not a legal cell in the grid, the character turns right:

```
if not LegalCell(testCell) then
  currentHeading := RightTurn(currentHeading);
```

In either case, the character moves to the next cell ahead.

```
currentCell := NextCell(currentCell, currentHeading);
```

Using the **else if** structure, when the character encounters a bomb or exit, the appropriate procedure (either Explode or MakeExit) is executed and the program terminates. Here, when the **if** condition is false, the **else** condition is evaluated; when the **if** condition is true, the **else** condition is skipped.

A Simple Strategy

We refer to this algorithm for proceeding through the maze as a “strategy.” Later on in the project, you’ll add other strategies that follow different algorithms.

Two Additional Procedures

The procedure **SetObstacle** is defined in the Graphics unit. It is a more general version of the procedure **SetBomb**, and can be used to draw any obstacle. It takes two arguments; the first is a cell and the second an integer that is assumed to represent an icon. For example, this statement draws a bomb in the current cell:

```
SetObstacle(currentCell, BOMB);
```

The **MakeExit** procedure is similar to the **Explode** procedure, except that it produces different animation and sound.



It takes two arguments—the cell in which the animation will appear and the icon ID of the character that has reached the exit. The **MakeExit** procedure is called when the character enters the cell containing the exit.

```
if SameCell(currentCell, exitCell) then
begin
    MakeExit(currentCell, icon);
    gameOver := true;
end;
```

Tinkering Initial Values

In this stage the Tinkering section describes the **Step Into Calls** command. It also suggests ways to alter the starting position of different characters, and outlines the process of identifying the various obstacle icons that are available.

Challenge: LeftTurn

Can you write a **LeftTurn** function? You might start by making a copy of **RightTurn**, then change the lines that need to be different. Remember that you need to complete both the interface part and the implementation part of a function declaration.

To take this one step further, try creating a more general **Turn** function that takes a single argument, assumed to be the direction of the turn (either left or right). To do this you might create a new enumerated type called **directionType**.

Stage Eight

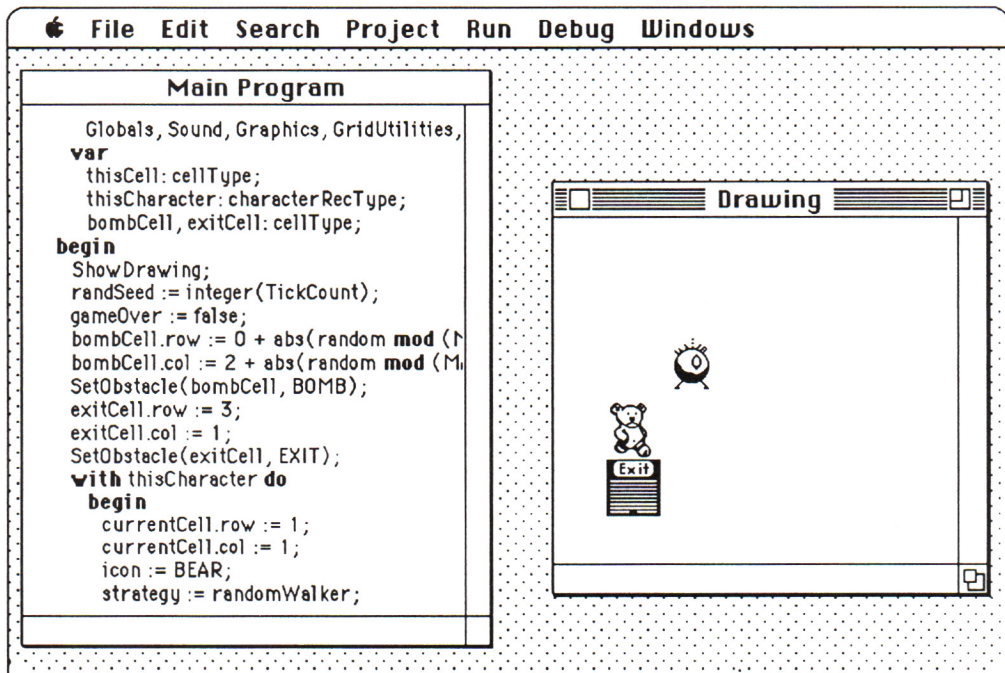
Random Numbers

Problem

The strategy that the character uses to run around the edge of the grid works nicely, as long as the exit is at the edge of the grid. If the exit is in the middle, the character will never find it. The characters need a strategy that is a bit more elaborate.

Solution: Random Numbers

In this stage, you assemble a strategy that uses the computer's random number generator. A character using this RandomWalker strategy will eventually find the exit (if it does not encounter a bomb first).



The bear using the RandomWalker strategy

Assembly

In Stage Eight you add:

- the Strategies Unit
- an enumerated type called `strategyType`
- a call to the function `RandomWalkerHeading`

Explanation Adding a Strategy

As you know, the algorithms that characters use to determine headings are called **strategies**. In this stage, you make strategy a formal part of the project.

A New Type

A new data type, `strategyType`, is declared in the `Globals` unit.

```
strategyType = (gridSkirter, randomWalker);
```

This enumerated type has only two possible values, corresponding to the two strategies in the project. Later, when you add your own strategies to the program, you can add other entries to the list.

The Strategy Field

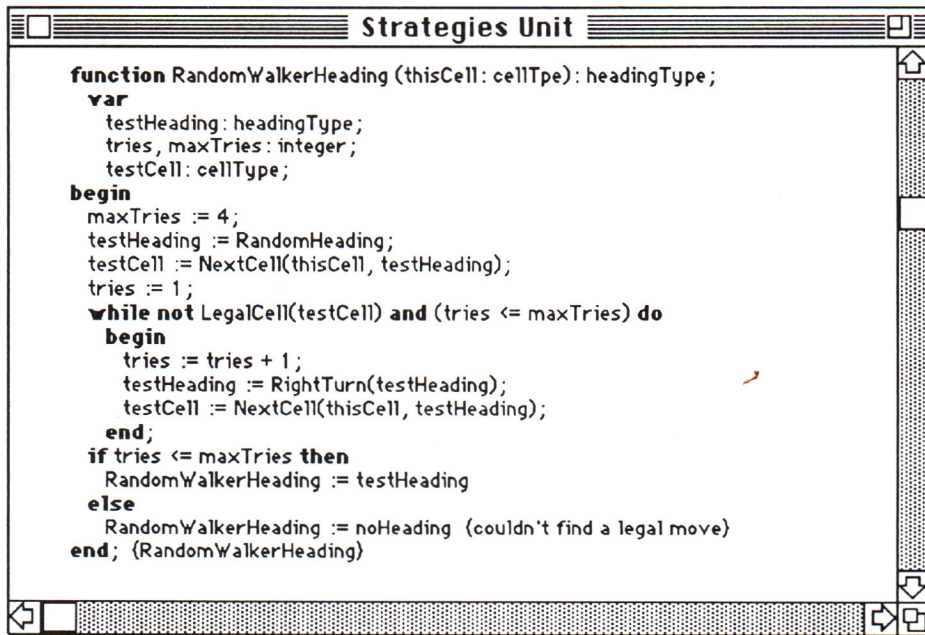
You also add a field called `strategy` to the declaration of `characterRecType` in the `Globals` unit.

```
characterRecType = record
    icon: integer;
    currentCell: cellType;
    currentHeading: headingType;
    strategy: strategyType;
end; {characterRecType declaration}
```

You can now use this to assign a strategy to a particular character. Later, when the program displays more than one character, different characters can follow different strategies.

RandomWalkerHeading

The function `RandomWalkerHeading` finds a random heading to a legal cell.



The RandomWalkerHeading function

Here is how it works:

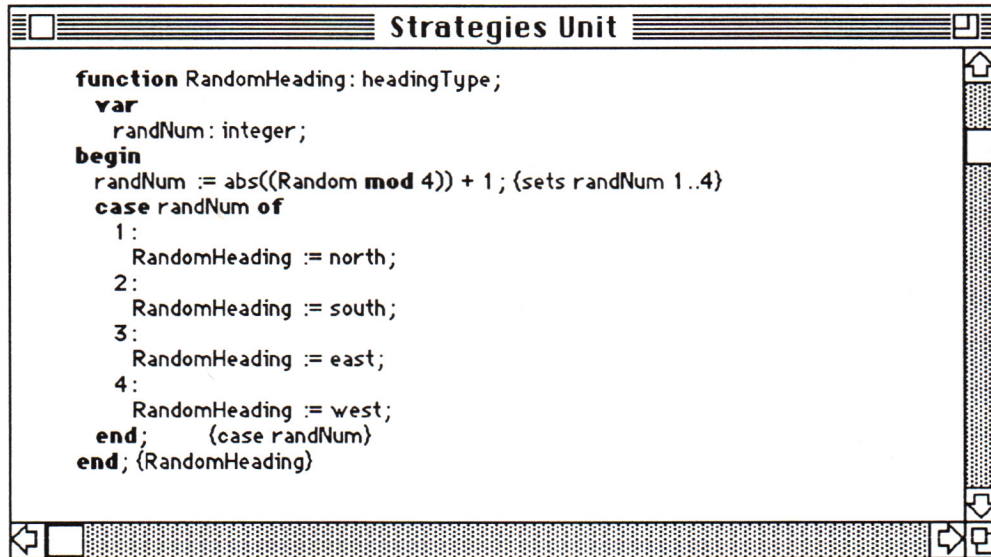
- The function first picks a heading (`testHeading`) at random (using the function `RandomHeading` described below) and checks to see if the next cell in that heading is legal (in the grid).
- If the cell is not legal, the function gets the next heading to the right and examines the next cell in that direction. After four tries, the function “gives up” and the loop terminates.
- The value of `RandomWalkerHeading` is set to `testHeading`, unless it has gone full circle without finding a heading, in which case `RandomWalkerHeading` takes the value `noHeading`.

JEP Note

You may wonder how `RandomWalkerHeading` could possibly *not* find a legal heading. While it is true that it will always find a legal heading at this stage, during later stages in the construction of the project it will be possible for a cell to be blocked on all four sides by obstacles or by other characters. That's when `noHeading` will come in handy.

RandomHeading

The function `RandomWalkerHeading` uses a separate function `RandomHeading` to choose randomly from the legal headings available. The statement that assigns a value to `randNum` is explained shortly.



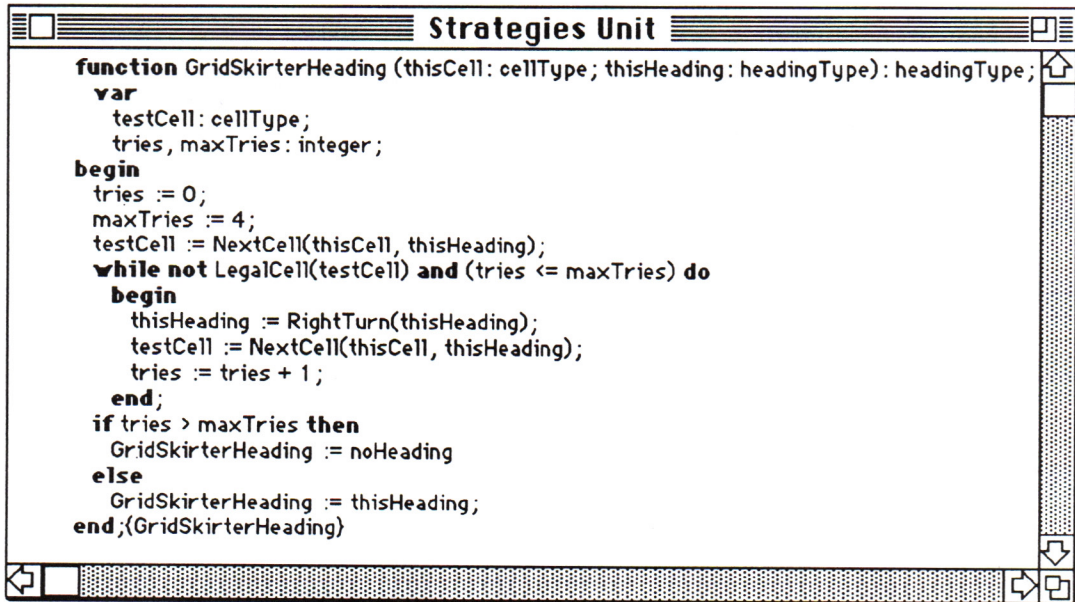
The RandomHeading function

JEP Note

You may wonder why there are two functions here—`RandomHeading` and `RandomWalkerHeading`—instead of just one. One reason is that you may want to use `RandomHeading` as part of another strategy. Generally speaking, it is a good idea to put separate tasks in different functions. This is called **modular** programming.

GridSkirterHeading

The function `GridSkirterHeading`, also in the Strategies unit, should look familiar. It consists of the statements that were previously part of the **repeat** loop in the main program rewritten as a separate function. You add statements to ensure that the function returns a heading to a legal cell.



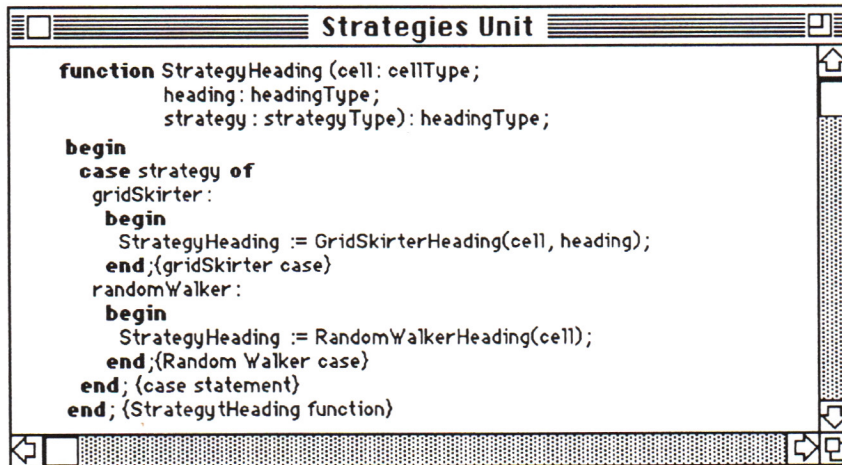
The GridSkirterHeading function

StrategyHeading

Finally, the main program calls the function StrategyHeading:

```
currentHeading :=  
    StrategyHeading(currentCell, currentHeading, strategy);
```

StrategyHeading in turn calls the appropriate strategy function, either GridSkirterHeading or RandomWalkerHeading. It returns the new heading for the character to the main program.



The function StrategyHeading

JEP Note

The use of a **case** statement in the StrategyHeading function will make it easy to add additional strategies when you create them later on. When you add another strategy, you'll be able to add another case to strategyHeading.

Changes to the main program

A single statement in the main program now gives the character a new heading that depends on its strategy.

```
currentHeading :=  
    StrategyHeading(currentCell, currentHeading, strategy);
```

Since this line is in a block headed by the line

```
with thisCharacter do
```

the arguments in the StrategyHeading function are actually fields in the variable thisCharacter. This statement calls the StrategyHeading function and passes the

currentCell, currentHeading, and strategy for thisCharacter. The statement then assigns the heading returned by StrategyHeading (which depends on the character's heading) to currentHeading. The heading returned by StrategyHeading depends on the character's strategy.

The **repeat** loop continues as before, moving the character into the new cell, erasing the old image of the character, drawing the character in its new position, and so on.

Random Numbers

RandomWalker generates headings that are randomly selected. Locate the implementation of the RandomHeading function and find the following line.

```
randNum := (abs(Random) mod 4) + 1;
```

This illustrates a compact way of creating a random number between 1 and 4.

mod

The keyword **mod** is a special operator that returns the remainder when the number preceding **mod** is divided by the number that follows it. For example, 7 **mod** 2 returns 1, because when 7 is divided by 2, the quotient is 3 and the remainder is 1. The result of 10 **mod** 5 is 0, because 5 divides 10 evenly.

Other operators include **div** (which divides two numbers and returns the whole number quotient) as well as *, /, + and -. These operators are part of the Pascal language and you can use them at any time.

Random

Random is an example of a function that takes no argument. It is a **QuickDraw** function that returns an integer between -32768 and 32767. The expression abs(Random) returns the absolute value of the random number, a number between 0 and 32767.

The expression abs(Random) **mod** 4 returns a random number between 0 and 3 because the remainder of any positive number divided by 4 is an integer between 0 and 3. Adding 1 to an integer between 0 and 3 produces an integer between 1 and 4.

Toolbox Note

QuickDraw is a set of commands that are part of the Macintosh Toolbox. You have access to these commands through the library Interface.lib. Consult the THINK Pascal *User's Manual* for details.

Seeding the random number generator

The QuickDraw function `Random` uses the Toolbox variable `randSeed` as a starting point for a sequence of random numbers. Changing `randSeed` (arbitrarily) produces a different sequence each time the program is run. When `randSeed` is not changed from one run of the program to the next, the same sequence is produced each time.

A statement near the beginning of the main program uses the Macintosh clock and the Toolbox function `TickCount` (another example of a function that takes no argument) to produce a different `randSeed` each time the program is run:

```
randSeed := integer(TickCount);
```

See the on-line Tinkering section for a continuation of this discussion of random numbers and `randSeed`.

Pascal Note

`TickCount` returns a `longint`. Passing this result to `integer()` converts the `longint` value into an `integer` value that can then be assigned to `randSeed`. This process of converting one type of value to another is called **casting**.

Tinkering Random Numbers

The Tinkering section of this stage suggests ways to experiment with the `Random` function using the Observe window.

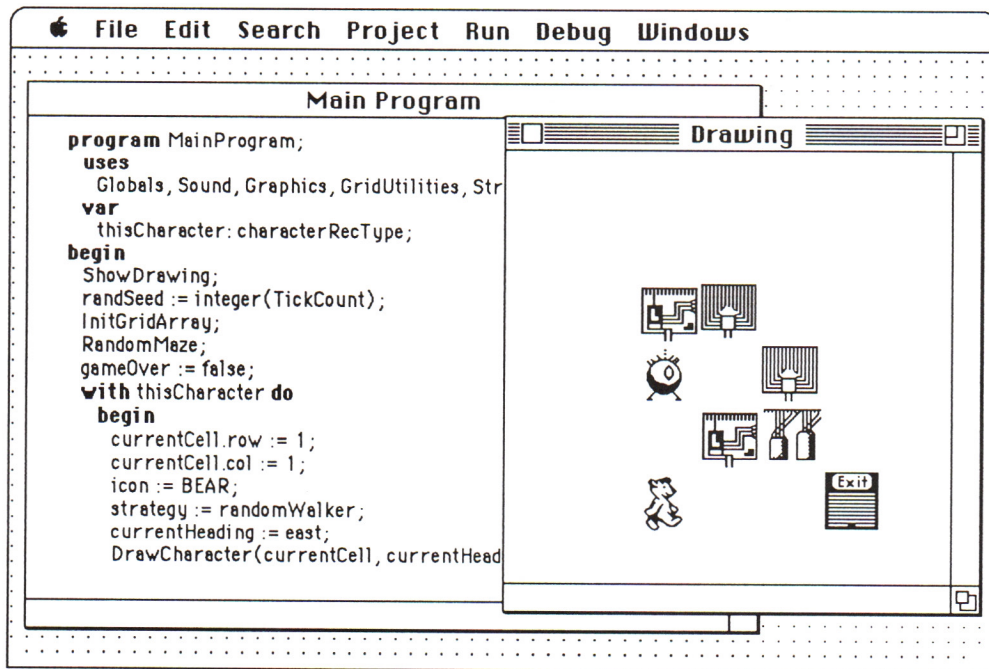
Stage Nine Arrays

Problem

Now you can position obstacles in the grid and move characters around the grid. However, the characters cannot distinguish between grid cells that have obstacles in them and grid cells that don't. You need a way to keep track of where the obstacles are.

Solution: Arrays

In this stage, you use an **array**—a special type of variable that works like a kind of table. To follow our analogy of variables as boxes, an array is a stack of boxes. Here you use a two-dimensional array, with rows and columns corresponding to the rows and columns in the grid. It's called `gridArray`.



The main program using gridArray

Assembly

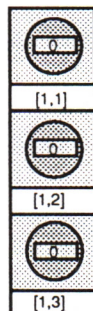
In Stage Nine you add to the project:

- the Array Management unit
- two new data types `gridCellType` and `gridArrayType`
- two functions `IsBombCell` and `IsExitCell`
- a small change to the definition of `LegalCell`
- changes to the main program

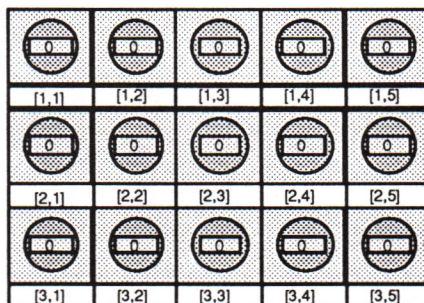
Explanation Arrays

This stage introduces a special kind of variable called an **array**. If a simple variable is a single box, then an array variable is a set of boxes stacked on top of each other.

Some arrays consist of a single stack of boxes; these are single-dimensional arrays.



Other arrays are two-dimensional; you can think of them as several stacks of boxes, forming a set of cubbyholes with rows and columns.



A three-dimensional array is also possible. Think of boxes stacked *behind* these boxes.

An array is an excellent means of storing information about the JEP grid, because both the array variable and the JEP grid can be viewed in terms of rows and columns.

Before you can create such an array, however, you must declare a new array type. Creating this array type is a three-step process. First, you define a special type for each cell in the grid (`gridCellType`). Then you define an array of those cells (`gridArrayType`). And finally, you declare a variable of the new array type (`gridArray`).

gridCellType

The new data type `gridCellType`, defined in the Globals unit, is a record that consists of a single field, `icon`, that is used to store the resource ID number of the icon that occupies a certain cell in the grid. This type will be used for the individual elements in the array.

```
gridCellType = record
    icon : integer;
end;
```

JEP Note

In this stage you could use a simple integer variable for `gridCellType` instead of a record. However, in a later stage you'll add another field to this record to keep track of a second fact about each cell—how many times a character has passed through it. This is another example of the flexibility of records.

gridArrayType

The array `gridArrayType`, defined in the Globals unit, is a two-dimensional array type in which the number of rows and columns in the array are determined by the global constants `MAXROWS` and `MAXCOLS`. Each element in this array is a variable of the type `gridCellType`. The type `gridArrayType` contains as many of these records as there are cells in the grid.

```
gridArrayType = array[1..MAXROWS, 1..MAXCOLS]
    of gridCellType;
```

When `MAXROWS` is set to 5 and `MAXCOLS` to 5, an array of `gridArrayType` consists of 25 elements. You might think of this array as a set of cubbyholes, 5 across and 5 high. Each element in the array is identified (**indexed**) by two numbers, one referring to the row and one to the column. The index of the element in the first row and first column is [1,1] and the index of the element at the corner diagonally opposite is [5,5].


























gridArray

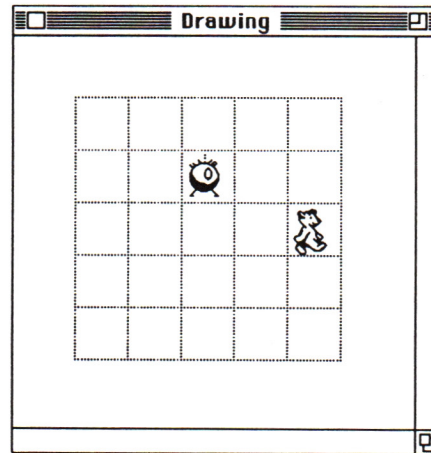
The variable `gridArray` is defined in the `Globals` unit:

```
gridArray: gridArrayType;
```

The `InitGridArray` procedure, discussed below, sets the initial values stored in `gridArray` to 0. You can now record the positions of characters and obstacles in the maze with assignment statements like these:

```
gridArray[2,3].icon := BOMB;  
gridArray[3,5].icon := BEAR;
```

				
.icon	.icon	.icon	.icon	.icon
[1,1]	[1,2]	[1,3]	[1,4]	[1,5]
				
.icon	.icon	.icon	.icon	.icon
[2,1]	[2,2]	[2,3]	[2,4]	[2,5]
				
.icon	.icon	.icon	.icon	.icon
[3,1]	[3,2]	[3,3]	[3,4]	[3,5]
				
.icon	.icon	.icon	.icon	.icon
[4,1]	[4,2]	[4,3]	[4,4]	[4,5]
				
.icon	.icon	.icon	.icon	.icon
[5,1]	[5,2]	[5,3]	[5,4]	[5,5]
gridArray				



JEP Note

Keep in mind the distinction between the assignment of a value to a cell in the grid and the procedure that draws a bear in that location. The assignment records the fact that the bear occupies that cell, but it does not draw the bear.

The Array Management Unit

The Array Management unit defines a variety of functions and procedures that are used in setting up and manipulating data stored in the different array variables that are used in the project. Three different arrays are used in the final project:

- `gridArray`—stores information about the grid
- `characterArray`—stores information about characters (Stage Ten)
- `paletteArray`—stores information about icons in the palette (Stage Thirteen)

Procedures For Handling Arrays

You now have four procedures in the Array Management unit related directly to `gridArray`:

- `InitGridArray`—fills the cells of the array with zeros
- `SetObstacle`—records, in `gridArray`, the position of an obstacle in the grid
- `IsBombCell` and `IsExitCell`—examine `gridArray` to locate either bombs or exits
- `LegalCell`—uses `gridArray` to determine if a cell is either empty or in the grid

InitGridArray

The procedure `InitGridArray` assigns 0 to the `icon` field of each cell in `gridArray`. The structure of the two **for** loops in this procedure is described later in this section.

Pascal Note

If `gridArray` was not initialized in this way, the values would not be 0, but arbitrary numbers like 59 or -31801. This is because Pascal does not automatically initialize variables to 0 (or any other specified value). The initial value for any variable is what happens to be in that memory location when the program starts.

SetObstacle

The variable `gridArray` has to be told which cells in the grid contain obstacles. A line added to the `SetObstacle` function, defined in the Graphics unit, handles this:

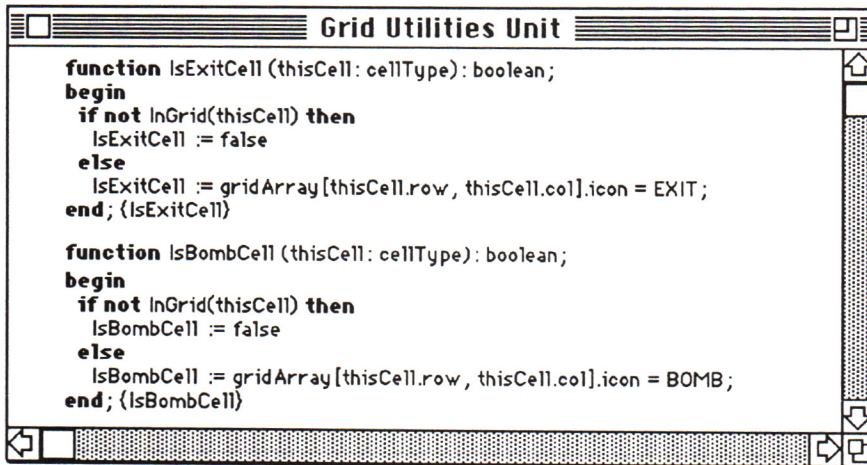
```
gridArray[thisCell.row,thisCell.col].icon := thisIcon;
```


With this change `SetObstacle` does two things:

- As before, `SetObstacle` draws an obstacle icon, specified by the variable `thisIcon`, in a cell specified by `thisCell`.
- `SetObstacle` also updates the new variable, `gridArray`, so that the icon field in the array cell corresponding to `thisCell.row` and `thisCell.col` is set to the value of `thisIcon`.

IsExitCell and IsBombCell

These two functions, defined in the Grid Utilities unit, determine if a given cell contains an exit sign or a bomb.



The functions IsExitCell and IsBombCell

Find this boolean expression in the function `IsExitCell`:

```
gridArray[thisCell.row, thisCell.col] = EXIT
```

This expression is true when the icon in the current cell of the grid is the EXIT icon; otherwise the expression is false. `IsExitCell` is assigned the value of this expression. `IsBombCell` works similarly.

If then else statements

Notice that these functions use the keyword **else** as part of the **if** statement. In an **if then else** statement, the expression following the **if** is evaluated. When this expression is true, the statement (or block) following **then** is executed. When the expression following **if** is not true, the statement (or block) following the keyword **else** is executed instead.

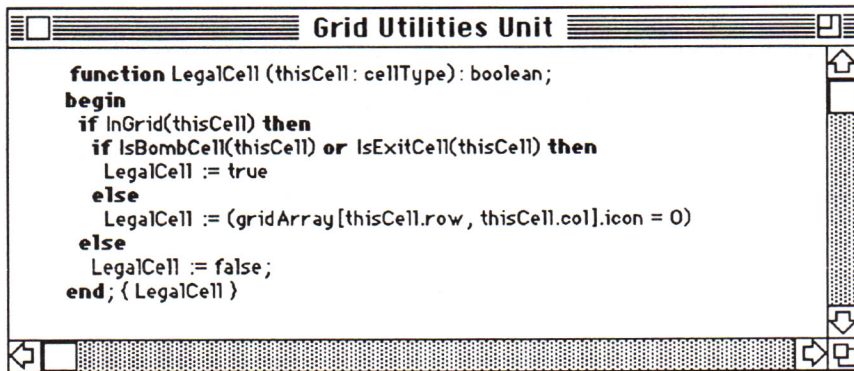
Notice that the expression following **if** includes the operator **not**. This operator negates the expression that follows it. For example, when `InGrid(thisCell)` is true, then **not** `InGrid(thisCell)` is false.

A more “intelligent” LegalCell

The function `LegalCell` in the Grid Utilities unit can now make use of information in `gridArray`. `LegalCell` returns true if the cell is in the grid (as determined by the function `InGrid`) and unoccupied (if the corresponding element in `gridArray` is 0).

In other words, a legal cell is one that a character can move into. It is inside the grid and it does not contain an obstacle. With this change to `LegalCell`, characters will avoid walking through obstacles as they negotiate a maze.

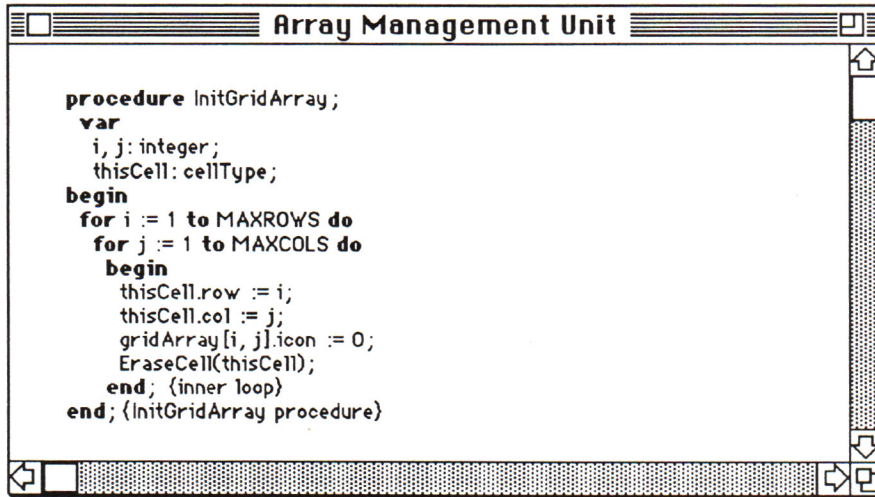
Notice that bombs and exits are treated as special cases.



The function LegalCell

Nested For Loops

The use of **for** loops in the procedure `InitGridArray` illustrates a useful programming technique called **nesting**. In this procedure two **for** loops are nested one inside the other, to initialize every cell in `gridArray`.



The function `InitGridArray`

First, the outer loop begins with this statement:

```
for i := 1 to MAXROWS do
```

The counting variable `i` is set to 1. The only statement in the program block for this loop is the statement that starts the inner **for** loop:

```
for j := 1 to MAXCOLS do
```

The second counting variable `j` is set to 1. At this point both `i` and `j` are 1. Next the block under the second **for** loop is executed. This statement assigns 0 to the `gridArray[1,1]`:

```
gridArray[i, j] := 0;
```

When the last statement in the inner loop is executed, `j` is incremented, and the block is executed again. This continues until `j` is greater than `MAXCOLS`. Think of the array as a table of cells—when the inner loop is finished, the first row (containing `MAXCOLS` cells) is filled with zeros.

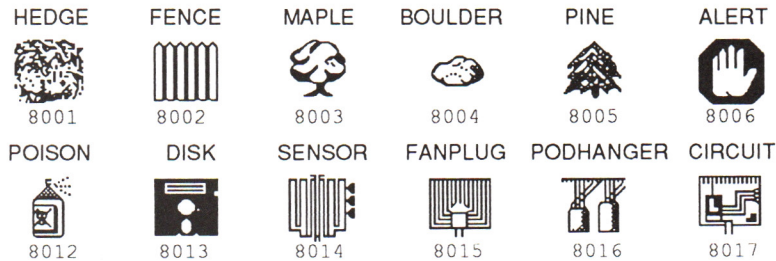
The process must be repeated for each row in the table of cells. When the inner loop finishes, the outer loop finishes too, because the outer loop consists of one compound statement—the inner loop. The counting variable `i` is incremented and the program block for the outer loop—the entire inner loop—is executed a second time. This fills the second row with zeros.

This continues until `i` is greater than `MAXROWS` and the outer loop is finished.

Tinkering RandomMaze

`RandomMaze`, a JEP procedure defined in the Array Management unit, draws several obstacles you haven't seen before. Like the other icons in the resource file, the resource IDs of these icons are declared as constants in the `Globals` unit.

The Tinkering section for this stage explores the operation of these two `for` loops using the `LightsBug` window.



Obstacle icons in the file `JEP Resources.rsrc`

The Tinkering section of JEP Instructions suggests ways to change this procedure to display these obstacles.

Challenge

Can you change the declaration of `RandomMaze` in such a way that it takes an argument specifying the number of obstacles to be drawn? For example, `RandomMaze (8)` might draw 8 obstacles, but `RandomMaze (4)` would only draw 4 obstacles.

Stage Ten

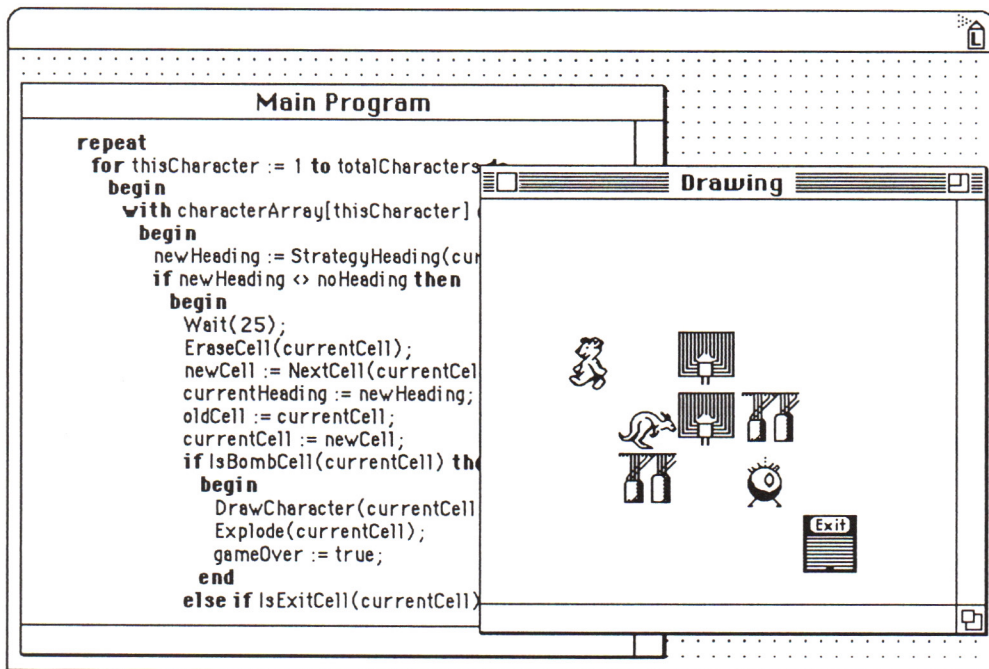
More About Arrays

Problem

Now you have two strategies, and a way to construct true mazes, but only one character. Wouldn't it be better if you could have more than one character in the maze at one time? That way, for example, you could assign different strategies to different characters, and then let them race to the exit.

Solution: The “Character” Array

In this stage, you set up an array of records (of `characterRecType`) to keep track of the status of several characters at once. You will also nest a **for** loop inside the original **repeat** loop. The inner loop will move individual characters.



GridWalker with multiple characters

Assembly

In Stage Ten you add:

- several small changes to the Globals unit
- a **for** loop to the main program
- a revised **repeat** loop

Explanation Changes to the Globals Unit

In the Globals unit, you declare a new constant, `MAXCHARACTERS`, a new data type, `characterArrayType`, and two new variables, `characterArray` and `totalCharacters`.

MAXCHARACTERS

`MAXCHARACTERS` is a constant that determines the maximum number of characters in the grid at any given time. At this stage, `MAXCHARACTERS` is set to 5. You can increase that if you wish, but bear in mind the following:

- `MAXCHARACTERS` is used to set the size of the array variable `characterArray`. When you increase `MAXCHARACTERS`, you will find that each character waits longer before it is animated.
- The variable `totalCharacters`, the total number of characters currently in the maze, cannot be greater than the constant `MAXCHARACTERS`.

characterArrayType

The new data type `characterArrayType` is an array of records:

```
characterArrayType = array[1..MAXCHARACTERS] of  
                    characterRecType;
```

This array is one-dimensional. Think of it as a single stack of cubbyholes. Each element in the array is a record, of the familiar `characterRecType`, which stores information about an individual character, including its current cell, heading, icon, and strategy.

characterArray

The variable `characterArray` is the only variable used of the type `characterArrayType`. Since it is declared in the Globals unit, which is used by every other unit, it is available to all procedures and functions in the program.

```
characterArray: characterArrayType;
```

A single index in square brackets following the name of the array variable identifies a particular element in the array. For example, `characterArray[2]` refers to the second record in the array.

1	1	east	3100	gridSkirter
.row	.col	.current-Heading	.current-lcon	.strategy
[1]				
2	2	east	3500	random Walker
.row	.col	.current-Heading	.current-lcon	.strategy
[2]				
characterArray				

totalCharacters

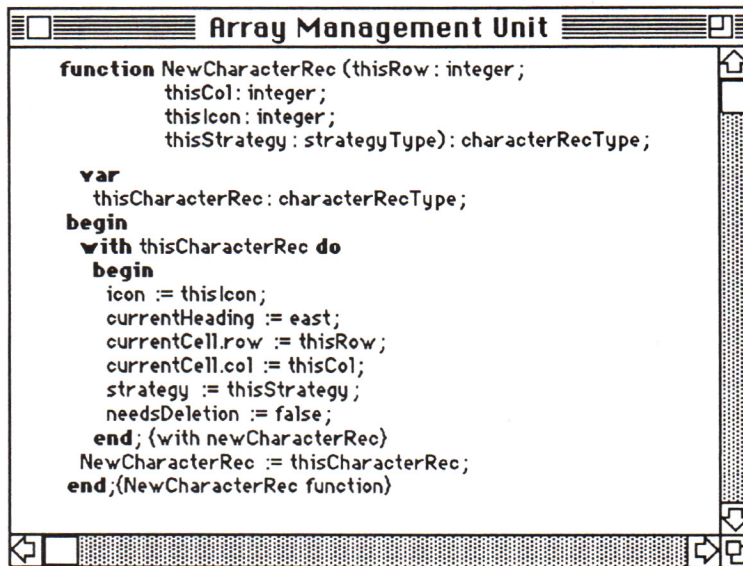
The integer variable `totalCharacters`, another global variable, is used to keep track of the total number of characters currently in the grid.

```
totalCharacters: integer;
```

The Function NewCharacterRec

The function `NewCharacterRec`, defined in the Array Management unit, takes three integers and a `strategyType` variable as arguments and returns a record of `characterRecType`. This function is used in the main program to create a record for a character.

`NewCharacterRec` declares a temporary variable `thisCharacterRec` and then makes all the necessary assignments to it using a **with** block. The last statement in the function assigns the contents of the temporary variable to `NewCharacterRec`. This technique is used because in defining a function in Pascal, only *one* assignment statement in the function can assign a value to the function name.



The function NewCharacterRec

Handling Multiple Characters

Originally the program drew a single character. Now it will draw several characters, using a **for** loop to march through the array of character records.

Four new variables are added to the main program:

- **thisCharacter**—an integer variable representing the index number of the character currently being drawn in the **for** loop
- **newHeading**—a headingType variable that records the change in heading as the character moves
- **oldCell**—a cellType variable that records the original position of the character being moved
- **newCell**—a cellType variable that records the position into which the character *will* move

Assignment to characterArray

The program begins by setting two characters in the grid. This initializing step assigns all the necessary information about these characters to the appropriate array variables. For example, the following statement creates the first character in characterArray.

```
characterArray[1] := NewCharacterRec(1, 1, BEAR, randomWalker);
```

The expression `characterArray[1]` designates the first element (a record) in the array. The character will appear in cell (1,1) as a bear that uses the RandomWalker strategy.

totalCharacters

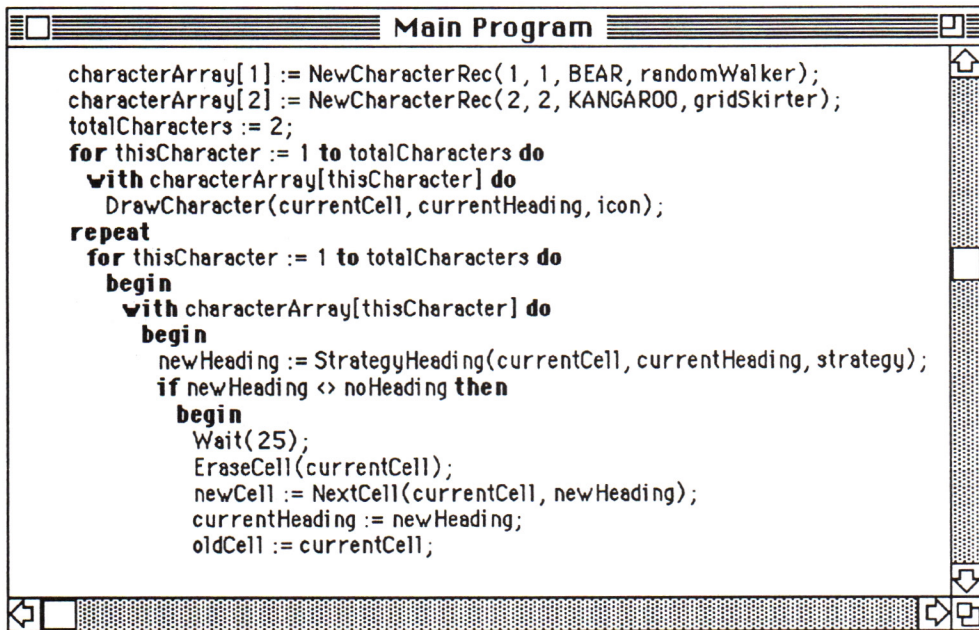
The integer `totalCharacters` is set to 2 in the main program, reflecting the fact that you are creating two characters. In later stages of the program, this variable will change as characters are created and deleted from `characterArray`.

JEP Note

If you create three new characters by adding an assignment statement to the main program, you must also change the assignment to `totalCharacters`.

For Loops in the Main Program

The main program now uses two **for** loops to draw all the characters in the `characterArray`. The first **for** loop, above the **repeat** loop, simply draws each character in its starting position. The second **for** loop, inside the **repeat** loop, executes several statements.



The two for loops in the main program

Both **for** loops set the counting variable `thisCharacter` to 1 and then execute the statements that follow. At the bottom of the loop, the variable `thisCharacter` is incremented and the statements are executed again. Both loops halt automatically when the

value of the counter variable exceeds the value of `totalCharacters`; that is, when `thisCharacter = totalCharacters + 1`.

Since the statements in the **for** loop all relate to a record in `characterArray`, the second statement in the loop is a **with** statement. Consequently, the variables are first interpreted as fields in the record of `characterArray` that has the index `thisCharacter`.

The second loop

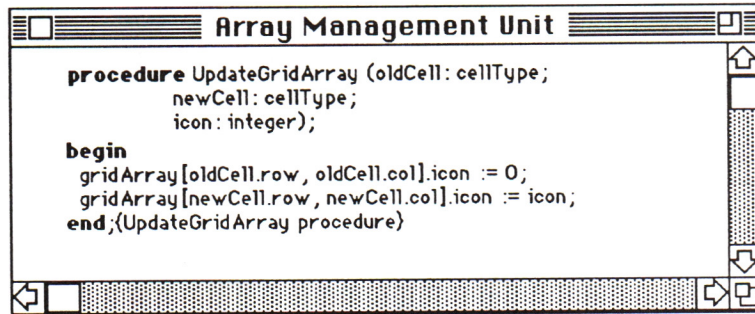
The second **for** loop is located inside the **repeat** loop. Several things happen inside this loop:

- `StrategyHeading` determines a new heading for the character.
- If the character cannot find a cell to move to, no new move is calculated.
- There is a slight pause, created by `Wait`.
- The character icon is erased from its current position.
- The program calculates which cell the character will move into based on the cell it is currently in and the new heading it will take.
- It records the old position of the character so that it can remove that character from `gridArray`, which keeps track of where the characters are.
- If the new cell contains a bomb or exit, the appropriate action is taken and the program stops.
- If the cell does not contain a bomb or exit, the character is drawn in the new cell.
- The change to the grid is recorded in `gridArray`, using the procedure `UpdateGridArray`.

The Procedure `UpdateGridArray`

The `UpdateGridArray` procedure, defined in the Array Management unit, informs `gridArray` when a character moves from one cell to the next. The procedure does two things:

- It erases information about the character's previous position by setting the appropriate element in the array to zero.
- It records information about the character's new position.



The procedure UpdateGridArray

Revisions to the Repeat Loop

The inner **for** loop is nested inside the **repeat** loop. On each pass through the **repeat** loop, the inner **for** loop is executed: it counts from 1 to `totalCharacters`, executing its block of statements once for each count.

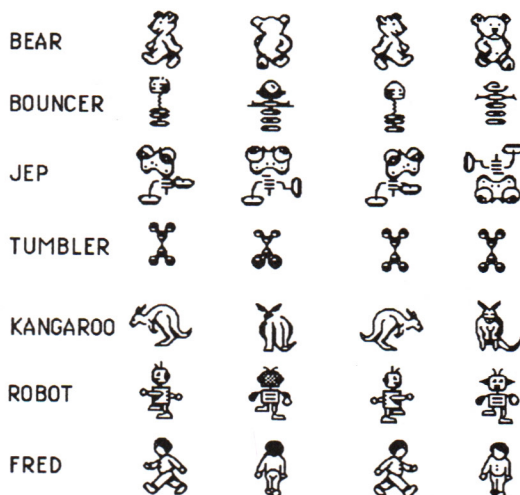
The **repeat** loop continues until `gameOver` is `true`. At that point the program halts.

JEP Note

You may find that the program does not halt immediately when a character runs into a bomb. This is because the **for** loop continues through the `characterArray` even after `gameOver` is set to `true`. If two characters hit bombs on the same pass through the **for** loop, both will explode before the program halts.

Tinkering

The Tinkering section outlines the steps you can follow to add a third character to the grid. It also suggests ways to experiment with the other characters that are available in the resource file.



The seven different characters

Challenge

Set up the program so that it runs *five* characters at the same time.

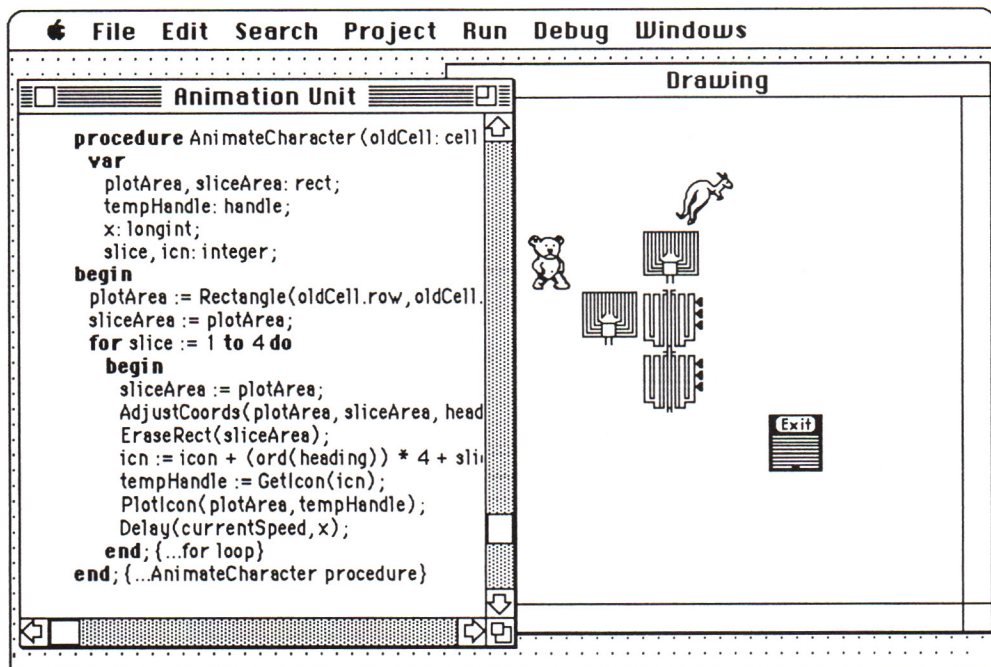
Stage Eleven Full Animation

Problem

Up to this point, the program uses a very simple animation technique. It draws an icon, erases it, and then draws it again in a different place. Although this gives the illusion of movement, the icons remain the same throughout the animation cycle, producing a rigid, jerky motion.

Solution: Improved Animation

In this stage, you use a more sophisticated kind of animation that uses several different icons for each animation cycle. Instead of a single icon showing a character facing west, you use four icons, each with the character's arms and legs in a different position. Using an erase, move, and draw cycle, you can make the character walk.



Using full animation in Stage Eleven

Assembly

In Stage Eleven you add:

- the Animation Unit
- the variable `currentSpeed`
- a call to the procedure `AnimateCharacter`

Explanation *Getting Technical*

The discussion in this section is provided for the benefit of those who need to know everything. If you're happy just using the new animation procedures, and don't need to understand exactly how the effect is created, you can safely skip over this Explanation and go straight to Tinkering. If you pass Go, you can still collect \$200.

The Animation Unit

The project now has an Animation unit. It comes with two procedures already defined.

- `AdjustCoords`
- `AnimateCharacter`

Character Icons

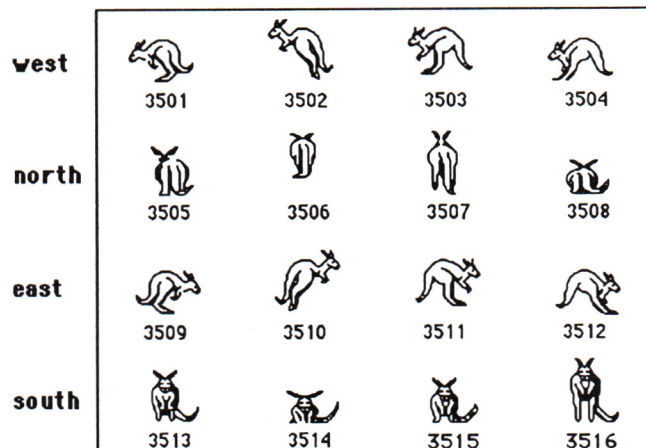
Each character has sixteen animation icons. The base number of these sixteen icons is taken from the character constant. For example, the constant `KANGAROO` is 3500, the resource ID for the kangaroo, so the sixteen animation icons range from 3501 to 3516. A list of the other character icons and their resource ID numbers is given in the Appendix.

Toolbox Note

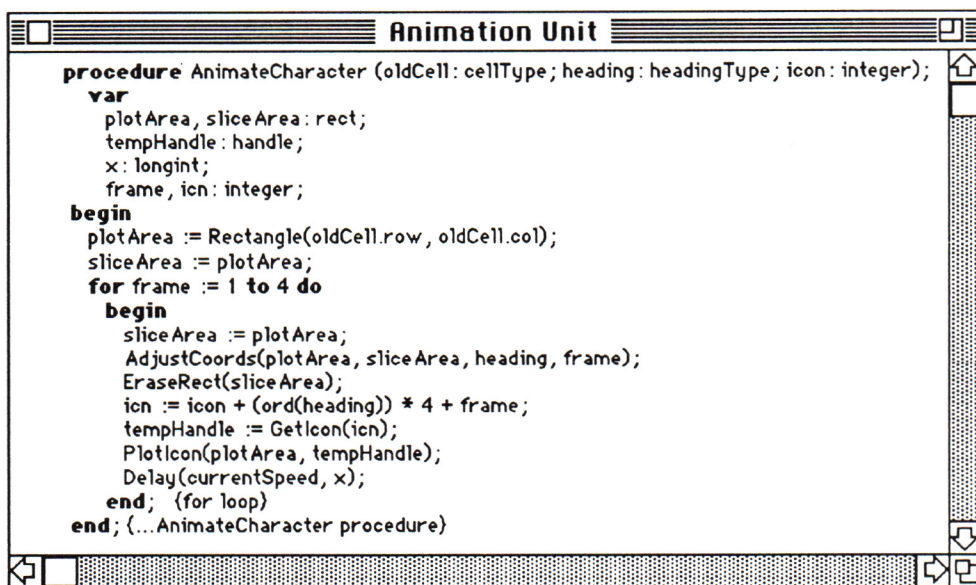
You can use the application `ResEdit` to examine the icons in the file `JEP Resources.rsrc` directly.

AnimateCharacter

The `DrawCharacter` procedure simply draws the character icon in a given cell with a given heading—the character jumps from one cell to the next. The `AnimateCharacter` procedure is more sophisticated—it draws the character four times as it moves from the old cell to the new cell. Each drawing shows the character in a slightly different pose, giving it the illusion of movement.



The sixteen Kangaroo icons



The AnimateCharacter procedure

Each character has 16 icons—four headings with four poses in each heading—and each icon has a specific resource ID. The procedure `AnimateCharacter` must select the correct number for a specific pose:

```
icn := icon + (ord(heading)) * 4 + frame;
```

The Pascal function `ord` returns the position number of an item in an enumerated list, the first item being 0. For example, the variable `heading` is taken from the list (`west`, `north`, `east`, `south`, `noHeading`). Therefore when `heading` is `west`, the expression `ord(heading)` returns 0 (because `west` is first in the list).

Suppose the character is in the third pose (the third “hop” of the kangaroo) as it moves north. The constant `KANGAROO` is 3500. Since `north` is second on the list of headings, `ord(north)` is 1. This number is multiplied by 4 because there are 4 icons for each heading. This part of the calculation stays constant through the four frames of the animation.

Finally, the value of the counter variable in the `for` loop (`frame`), which is now 3, is added to the expression. This selects the third icon. Finally, the variable `icn` is assigned the total value.

```
icn := icon + (ord(heading))*4 + frame;  
icn := 3500 + (1 * 4) + 3
```

Check the kangaroo icons above to see that 3507 shows the kangaroo in the third hop, facing north (its back to the viewer). Four frames make up the complete animation, with the last frame leading back into the first.

AdjustCoords

The `AdjustCoords` procedure, also in the `Animation` unit, determines the position of the rectangular area that must be erased as the character moves across the screen. This rectangle depends on the current heading of the character and its position within the cell. A `case` statement is used to handle each of the four directions. This procedure makes the animation smoother because only a portion of the area filled by the icon is actually erased.

Setting the Speed of the Animation

The following line sets the initial speed of the animation.

```
currentSpeed := 5;
```

The global variable `currentSpeed` is used in this statement in `AnimateCharacter`:

```
Delay(currentSpeed, x);
```

`Delay` is a Toolbox procedure that pauses the program for a number of ticks equal to `currentSpeed` (a tick is 1/60 of a second). To make the animation run faster, set the `currentSpeed` variable to a smaller integer. The variable `x` returns the length of time in ticks since the last call to `Delay`. The program does not use this piece of information, but the parameter must be included so that the parameter list matches that of the procedure definition in the Toolbox.

Changes to the Main Program

You can now replace the `DrawCharacter` procedure in the main program with the `AnimateCharacter` procedure. The arguments are the same. The result is that instead of characters hopping from cell to cell, they animate smoothly between cells.

Tinkering

The Tinkering section suggests experiments you can conduct to see the different characters available in the resource file move around the screen. You can also tinker with the speed of the animation.

Challenge

Can you make different characters move at different speeds? You might approach this challenge by adding a new field to `characterRecType`.

Congratulations!

You've covered quite a bit of ground. This is perhaps a good place to take a break and consider what you have done so far. First, the program you've constructed is already quite elaborate:

- *Using `SetObstacle` and `RandomMaze` you can construct a variety of new mazes in the Drawing window.*
- *You can have several different characters work their way through your mazes at the same time, following one of two basic strategies.*
- *And you can animate those characters, so that they appear to walk, or in the case of the Kangaroo, hop.*

*You've also explored many terms and concepts of Pascal programming. You've used variables, records, arrays, **if** statements, **for** loops, **repeat** loops, units, procedures and functions. These first eleven assembly stages constitute much of the content of an introductory course in Pascal programming. (Things we haven't covered include strings, pointers, and sets.)*

In the remaining stages, you will work closely with the functions and procedures that are specific to the Macintosh Toolbox—these are the tools that let you create menus and windows and let your program respond to actions on the part of the user.

Stage Twelve

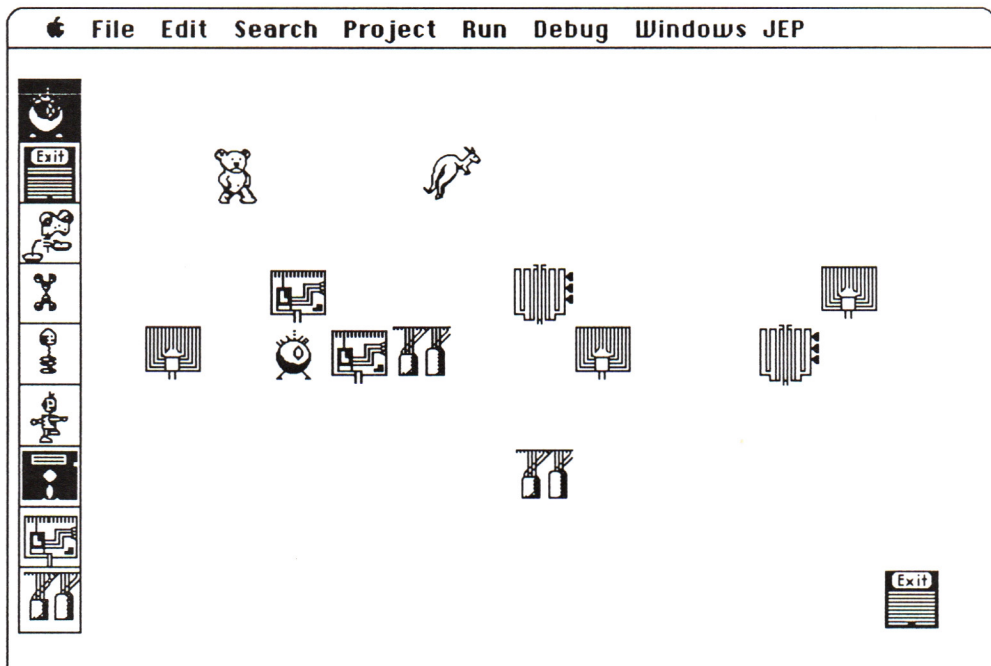
The Palette

Problem

Up to this point in the assembly, you, the programmer, are the only one who can control how your program behaves. For example, you could plant several bombs in the maze, but you would have to change the program itself to do so. And only *you* know how to do that.

Solution: The User Interface

If you want others to be able to use your program, you must build what is called a “user interface.”



Drawing the palette in Stage Twelve

The User Interface

The user interface we have in mind includes:

- A palette, drawn down the left side of the screen, to display all the character and obstacle icons, much like the palette in MacPaint™ displays drawing tools.
- The Macintosh menu bar, along the top of the screen, with standard File and Edit menus, as well as special menus that set the animation speed and change strategies.
- A main event loop, which responds appropriately when the user clicks the mouse.

In this and other ways, you will turn full control of the program over to the user. This, after all, is the ultimate goal of most programming projects.

Assembly

In Stage Twelve you add:

- the Palette unit
- a new data type called `paletteArray`
- a call to the function `InitPalette`

Explanation The Palette

The JEP palette is a display of icons (representing both obstacles and characters) that is drawn down the left edge of the Drawing window. By Stage Sixteen, when the user clicks on an icon in the palette, that icon will be selected and can then be added to the grid.

To create the palette, you make the grid larger, create a new array, and add a unit to the project. When you're finished with this stage, the palette will appear on the screen, but users won't be able to use it yet.

Adding constants

The following constants, defined in the `Globals` unit, help to identify the different parts of the screen.

```
PALETTE = 1;  
GRID = 2;  
NONE = 0;
```

Making the Grid Larger

To make room for the palette, expand the size of the grid to 9 rows and 15 columns by changing the constants `MAXROWS` and `MAXCOLS` in the `Globals` unit. All operations that use the constants `MAXROWS` and `MAXCOLS` will change automatically. For example, `gridArray` will consist of 135 cells instead of 25. The function `InGrid` will recognize the larger grid, and so on. This is a good illustration of the usefulness of constants.

The variable `paletteArray`

The global variable `paletteArray` is an array of `MAXROWS` integers that correspond to the resource IDs of the palette icons. You can have as many icons in the palette as there are rows in the grid.

```
paletteArray: array[1..MAXROWS] of integer;
```

The Palette Utilities Unit

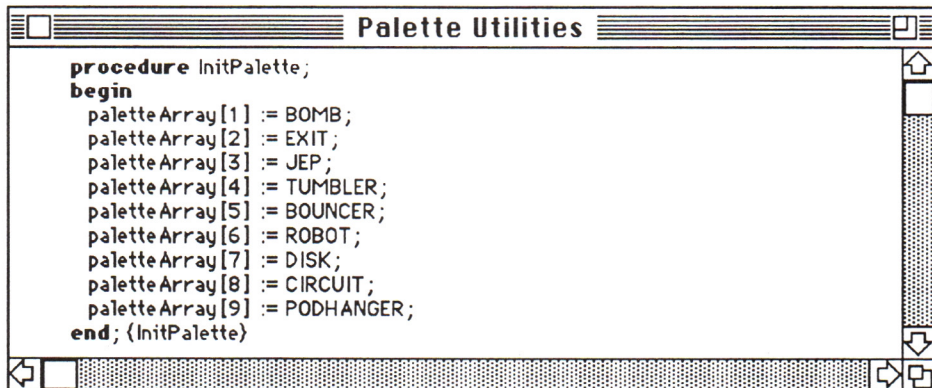
The `Palette Utilities` unit contains three procedures that handle basic palette operations:

- `InitPalette`
- `DrawPaletteIcon`
- `DrawPalette`

`InitPalette`

The `InitPalette` procedure assigns values to each of the elements in `paletteArray`. For example, the following line assigns to the first element in `paletteArray` the value of the constant `BOMB`, which is defined in the `Globals` unit as 8020.

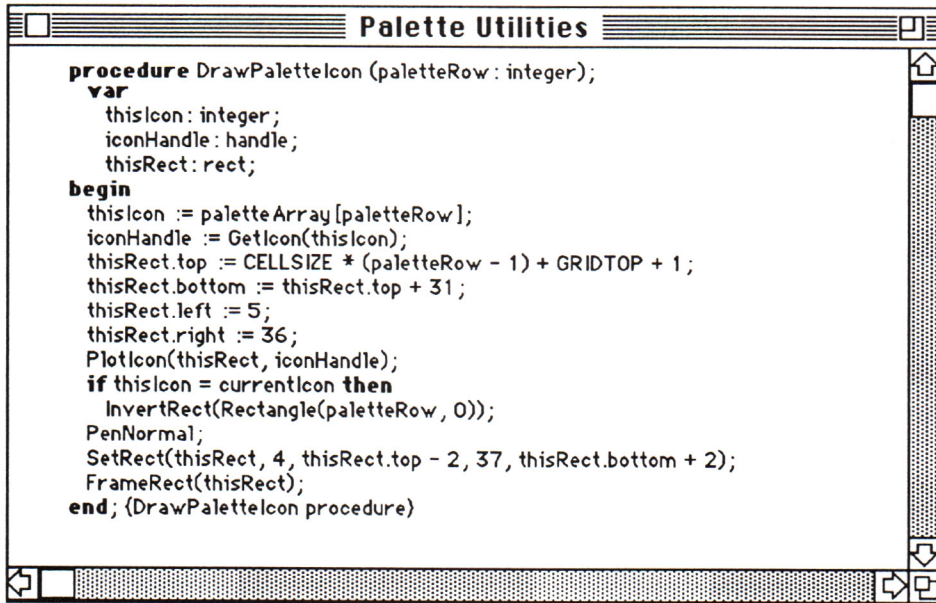
```
paletteArray[1] := BOMB;
```



The procedure `InitPalette`

DrawPalettelcon

The DrawPalettelcon procedure takes a single argument, an integer that corresponds to a row in the palette, and draws the appropriate icon in that row.



The procedure DrawPalettelcon

These statements at the end of the procedure outline the icon:

```
SetRect (thisRect, 3, thisRect.top - 2, thisRect.bottom + 2);
FrameRect (thisRect);
```

Both SetRect and FrameRect are QuickDraw procedures. The first statement sets a drawing rectangle that is slightly larger than the icon. The second statement frames that rectangle with a black line.

Examine this statement in the procedure:

```
if thisIcon = currentIcon then
  InvertRect (Rectangle (paletteRow, 0));
```

This statement is executed as follows: When the condition evaluates as true (that is, when the icon being drawn, thisIcon, is the same as the currently selected icon, currentIcon), then the function Rectangle is called with the arguments paletteRow and 0 (0 is the left-most column); Rectangle returns a value of type rect that corresponds

to a cell in the palette. The QuickDraw procedure `InvertRect` inverses the pixels in that region (i.e., it makes the black pixels white and the white pixels black).

The inverted cell in the palette shows the user which icon is currently selected.

DrawPalette

The `DrawPalette` procedure is a simple `for` loop that calls `DrawPaletteIcon` once for each row in the palette. Defining `DrawPaletteIcon` separately from `DrawPalette` makes it possible to draw palette icons individually. This will prove useful later when the program inverts an icon on the palette when the user selects it.

Making the Drawing Window Larger

To make room for the larger grid, you need to enlarge the Drawing window so that it fills the screen. You add these statements to the main program:

```
SetRect (fullScreen, 0, 0, 560, 350);
SetDrawingRect (fullScreen);
```

The first statement defines the size and position of a QuickDraw rectangle (a variable of type `rect` called `fullScreen`) with the integers 0, 0, 560, and 350. This rectangle begins in the upper left corner of the window (0,0) and extends over 560 pixels and down 350 pixels. The THINK Pascal procedure `SetDrawingRect`, defined in the library `Runtime.lib`, displays the Drawing window in this rectangle.

Toolbox Notes

QuickDraw uses the `rect` type in several different ways. Here it is used to define the size and position of a window. In the palette, the variable `thisRect` determines where lines are drawn. Several procedures in the Graphics unit use variables of type `rect` to plot icons in the grid with the QuickDraw procedure `PlotIcon`.

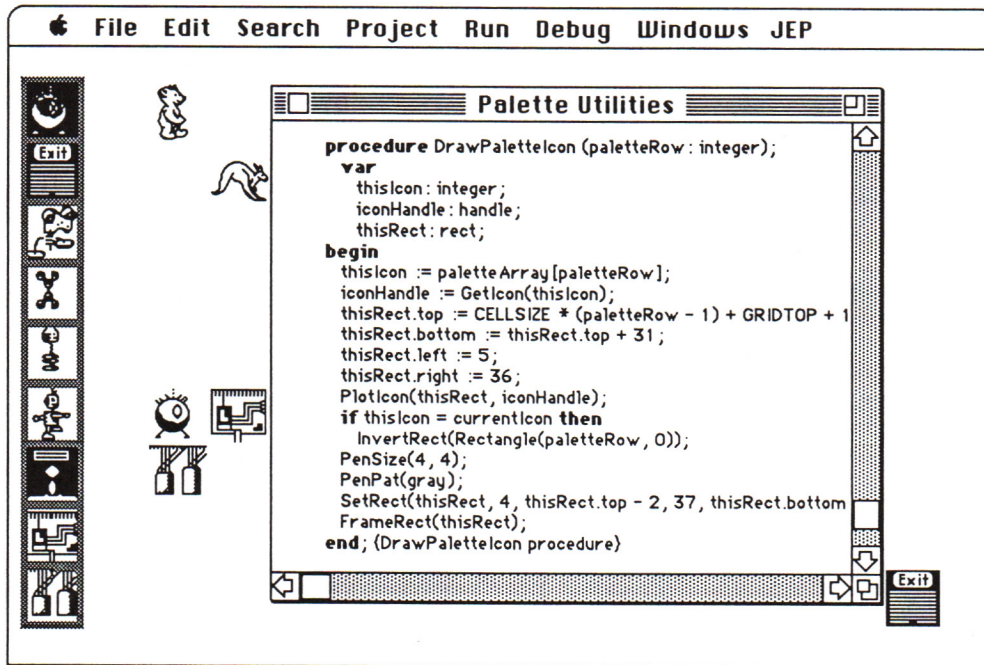
JEP Note

The Macintosh screen is actually 512 pixels by 342 pixels. The program uses slightly larger values here to make the window larger than the screen. As a result, the resize box, title bar, and scroll box do not show.

The Drawing window now fills the screen, hiding the JEP Instructions window and the GridWalker project and edit windows. When the Drawing window is active, choose **Close** from the File menu to close it and make the other windows visible again.

Tinkering QuickDraw

The Tinkering section suggests several experiments with QuickDraw that change the thickness and pattern of the lines drawn around the border of the palette.



Palette drawn with a different border

Challenge

Alter `InitPalette` in such a way that it draws a selection of icons that are *randomly* selected from the total set of available icons. See the Appendix for a list of all icons available in the file `JEP Resources.rsrc`.

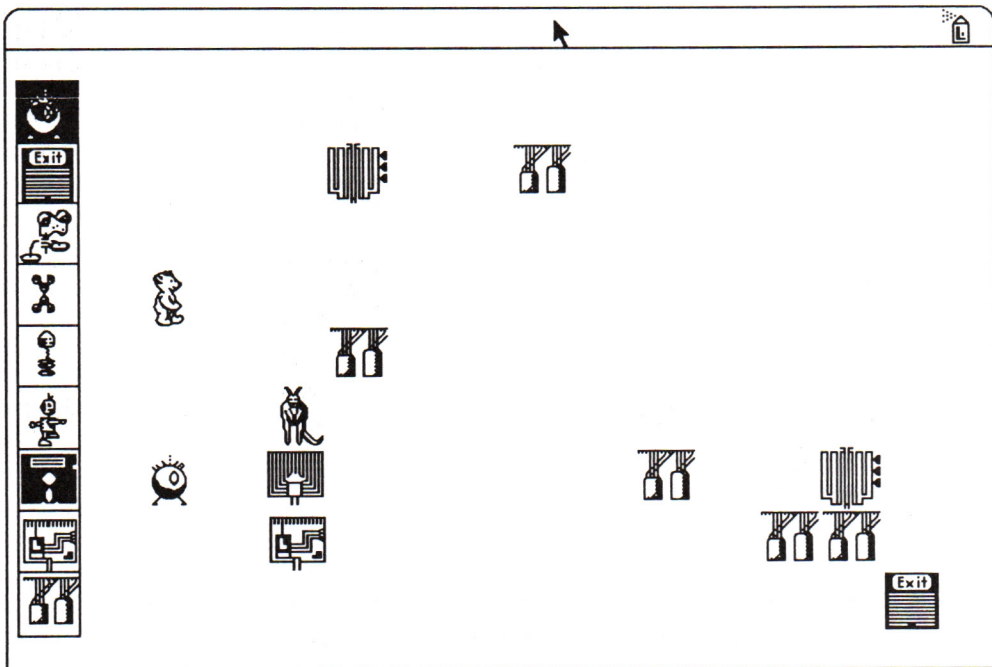
Stage Thirteen

The Main Event Loop

Problem

You constructed the palette in the previous stage, but it's not working yet. For example,

- When the user clicks the mouse with the pointer in the palette area, the program should select the icon under the pointer by inverting that icon and uninverting (returning to normal) the icon selected previously.
- If the user clicks the mouse with the pointer in an empty cell in the grid, the selected icon should be placed there.
- If the user clicks in a cell in the grid that is not empty, the contents of the cell should be erased.



Clicking in the menu bar to start the animation

Solution: The Main Event Loop

To make the program work this way, you'll have to keep track of when and where the mouse is clicked. This will involve adding what programmers call a "main event loop" to the program.

When you complete this stage, the user will be able to start and stop the animation by clicking the mouse with the pointer in the menu bar.

Assembly

In Stage Thirteen you add:

- a new **repeat** loop in the main program (the main event loop)
- the procedure `MoveCharacter`

Explanation The Main Event Loop

In this stage we add a main event loop to the main program. This loop is a **repeat** loop that continues until `gameOver` is `true`. Everything else the program does—drawing characters, finding their next moves, updating arrays, generating sounds, and so on—happens inside this loop.

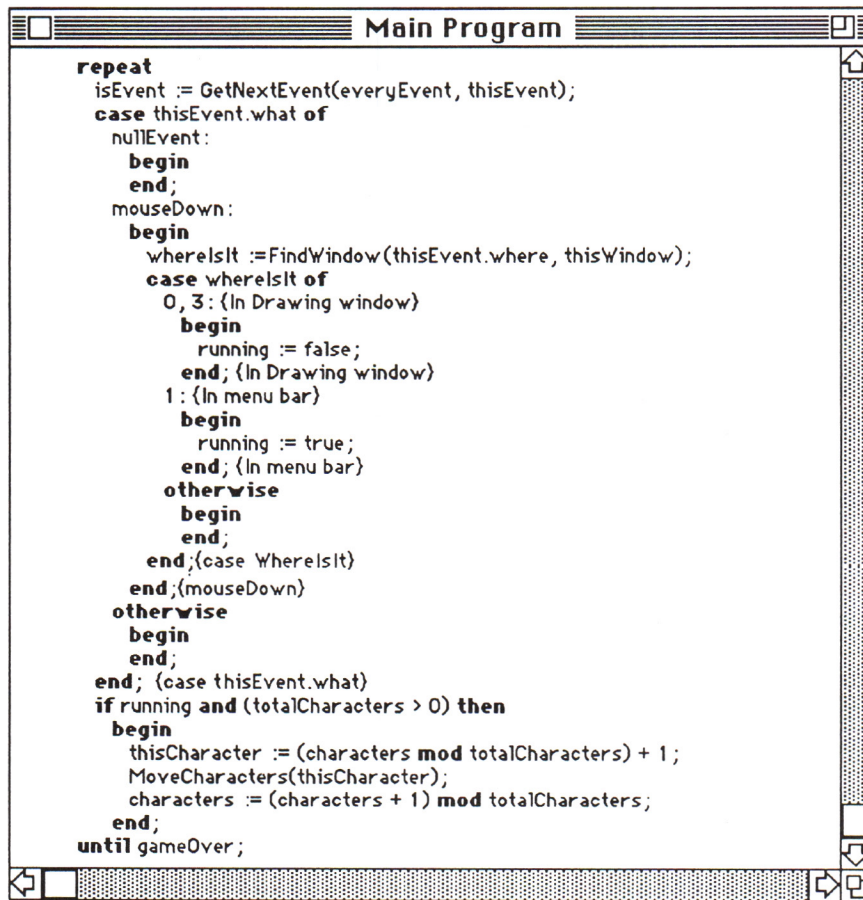
The main event loop follows this logical sequence:

- Check if the user has done anything that must be dealt with.
- If so, respond to what the user has done.
- If not, move the next character, update arrays, make sounds, etc.
- Repeat.

In other words, the program goes about its business as usual, always on the lookout for user actions such as a mouse click.

Toolbox Note

A "mouse click" is more formally known as a **mousedown event**. When we talk about a mouse click in a certain location, such as the menu bar, we mean that the user has clicked the mouse with the pointer in that location.



The main event loop

The Function GetNextEvent

GetNextEvent is a Macintosh Toolbox function that determines if the user has clicked the mouse, and, if so, where the pointer was.

GetNextEvent is formally defined as follows:

```

function GetNextEvent (eventMask: integer;
                       var theEvent: eventRecord ): boolean;

```


eventMask

The first parameter, `eventmask`, determines what kinds of events you want the main event loop to be on the lookout for. For example, you might only look for events in which the mouse button is pushed down. Or you can look for every event. In fact, the `ToolBox` constant `everyEvent` instructs `GetNextEvent` to report every event, including mouse clicks, keyboard presses, or disk insertions. These events are recorded in the event queue, described below.

The `ToolBox` defines constants as well as functions and procedures. The constants `everyEvent`, `mouseDown`, `nullEvent`, and so on, are also discussed below.

Event Record

The variable type `eventRecord` contains information about the event that has been detected. The `eventRecord` includes these fields:

```
type eventRecord = record
    what: integer; {event code}
    where: point;   {mouse location}
```

For a complete definition of an `eventRecord`, see *Inside Macintosh*, I-249.

what

The `what` field of an `EventRecord` is represented by `Toolbox` constants. Two are important to our program:

```
nullEvent  = 0;
mouseDown  = 1;
```

If the `what` field contains 0, nothing has happened. If it contains 1, the mouse was clicked.

where

The `where` field of an `EventRecord` is defined as a `point`—a record with two fields:

- the horizontal position of the mouse (`h`)
- the vertical position of the mouse (`v`)

theEvent

The variable `theEvent`, an `eventRecord`, keeps track of all this information. The keyword **`var`** in the parameter list marks it as a **variable parameter**. That means that the information assigned to that variable can be changed by the function `GetNextEvent` when it is called. `GridWalker` does not use **`var`** parameters in this way.

Using GetNextEvent

All this information about an event is recorded and made available to you as a programmer each time the program calls the `GetNextEvent` function:

```
isEvent := GetNextEvent(everyEvent, thisEvent);
```

In this statement, the function `GetNextEvent` fills the record `thisEvent` with information about the first event that occurred since you last called it and assigns to the boolean variable `isEvent` the value `true`. When an event has not been detected since the last time `GetNextEvent` was called, `isEvent` is set to `false`.

Events are recorded in a **FIFO** (first-in, first-out) event queue. This event queue is like a stack of checkers; each event is a checker added to the top of the stack. When you call the `GetNextEvent` function, the bottom checker on the stack is removed from the stack and analyzed.

If the mouse click occurs while the program is busy doing something else, like drawing a character, the event is still recorded in the event queue. You don't have to worry about calling `GetNextEvent` at the exact moment the mouse click occurs. When you call `GetNextEvent`, it retrieves that event automatically from the queue.

Creating the Main Event Loop

Find the main event loop in the main program. It begins with a **repeat** loop:

```
repeat
    isEvent := GetNextEvent(everyEvent, thisEvent);
    .
    .
until gameOver;
```

On each pass through the loop, the first statement takes the last event from the event queue and assigns the information about that event to `thisEvent`, an `eventRecord`.

For example, suppose `thisEvent` is a mouse click 20 pixels from the left edge of the screen and 50 pixels down. This event produces the following results:

• <code>thisEvent.what</code>	<code>mouseDown</code>
• <code>thisEvent.where.h</code>	20
• <code>thisEvent.where.v</code>	50

Again, `mouseDown` is a Toolbox constant that is defined as 1.

Case thisEvent.what

The main event loop then uses a **case** statement to handle the different events:

```
case thisEvent.what of
  nullEvent:
    begin
    end;
  mouseDown:
    begin
      .
      .
      .
    end; {mouseDown}
  otherwise:
    begin
    end;
end; {case thisEvent.what}
```

If no events were placed in the queue since the last call to `GetNextEvent`, then the `nullEvent` case is satisfied and no actions are taken. Control passes out of the **case** statement to the statements below it.

If `thisEvent.what = mouseDown`, then the block of statements under the `mouseDown` case is executed. All other events are handled by the **otherwise** case. Like the `nullEvent` case, no statements are listed under the **otherwise** case; when this case is executed, no actions are taken.

FindWindow

When a `mouseDown` event is detected, the program must decide where it occurred. The first statement in the `mouseDown` case is:

```
whereIsIt := FindWindow(thisEvent.where, thisWindow);
```

`FindWindow` is a Toolbox function that determines which part of the window the pointer was in when the mouse button was pressed. It is defined as:

```
function FindWindow (thePt: Point;
                    var whichWindow: WindowPtr ): integer;
```

`FindWindow` takes two arguments, a point and a window pointer. The parameter `thisEvent.where` is assigned to the point argument. `FindWindow` returns an integer that corresponds to the constants:

<code>inDesk</code>	0
<code>inMenuBar</code>	1
<code>inContent</code>	3

In other words, if `FindWindow` returns a value of 1, the user clicked in the menu bar. If `FindWindow` returns a value of 0 or 3, the user clicked in the Drawing window. We can ignore the variable `whichWindow` because `GridWalker` uses only one window.

Toolbox Note

A window pointer is a special Toolbox type. See *Inside Macintosh* I-275 for details.

Case whereIsIt

The program uses a second **case** statement to deal with mouse clicks in different locations:

```
whereIsIt := FindWindow (thisEvent.where, thisWindow);
case whereIsIt of
  0,3 : {in Drawing Window}
    begin
      running := false;
    end; {In Drawing window}
  1 : {In menu bar}
    begin
      running := true;
    end; {In menu bar}
  otherwise
    begin
    end
end; {case WhereIsIt}
```

If the pointer is in the Drawing window when the mouse is clicked, the animation stops running. If it is in the menu bar, the animation starts.

In later stages you'll have a chance to elaborate your responses to these mouse clicks, but the basic structure of the main event loop remains the same.

Finishing the Loop

After the **case** `thisEvent.what` statement, the program continues with this statement block:

```
if running and (totalCharacters > 0) then
  begin
    thisCharacter = (characterCount mod totalCharacters) + 1;
    MoveCharacter(thisCharacter);
    characterCount := (characterCount + 1) mod totalCharacters;
```


If the animation is running (the boolean variable `running` is `true`) and there is at least one character in the grid (`totalCharacters > 0`) then the `MoveCharacter` procedure in the `Animation` unit is called with the counting integer `thisCharacter`. Each time through the main event loop, one character is animated.

The boolean variable `running` is useful because it lets the user temporarily halt the movement of the characters without stopping the program.

The **repeat** loop continues until the boolean variable `gameOver` is `true`. This variable is changed when a character hits a bomb or an exit.

MoveCharacter

The procedure `MoveCharacter`, added to the `Animation` unit, is based on the **for** loop we were using in the main program in the previous stage. Taking it out of the main program greatly simplifies the main program and makes it easier to work on the main event loop.

The `MoveCharacter` procedure moves one character in `CharacterArray`—it finds a legal move, animates the character, and updates `gridArray` to reflect the character's new position in the grid.

Tinkering

The Tinkering section of the JEP Instructions explores the main event loop.

For example, by following these instructions, you can see that a mouse click in the lower left corner of the Drawing window produces the value `true` for the variable `isEvent` and the value 1 (a mousedown event) for the variable `thisEvent.what`. Also, the location of the mouse click is recorded in `thisEvent.where.h` and `thisEvent.where.v`.

Challenge

Add statements to the main program so that the program beeps when the user clicks in the menu bar. Use the procedure `SysBeep(10)`.

123

Assembly

In Stage Fourteen you add:

- two variables to the main program
- four new functions to the Grid Utilities unit

Explanation

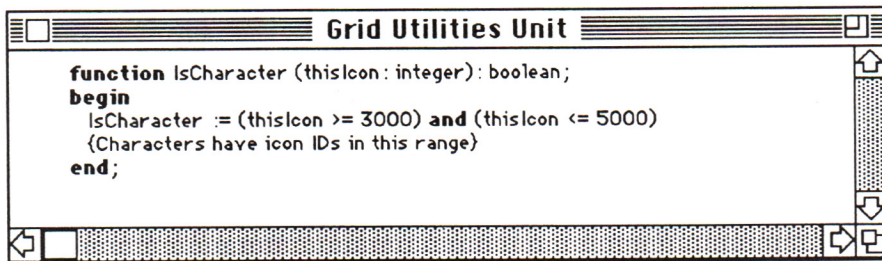
Responding to Mouse Clicks

In Stage Fourteen the program drew the palette on the screen and detected clicks of the mouse. In this stage you put the two together to detect mouse clicks in the palette. To do this, you add four functions to the Grid Utilities unit:

```
IsCharacter  
IsObstacle  
ClickCell  
EmptyCell
```

IsCharacter

This function is defined in the Grid Utilities unit.

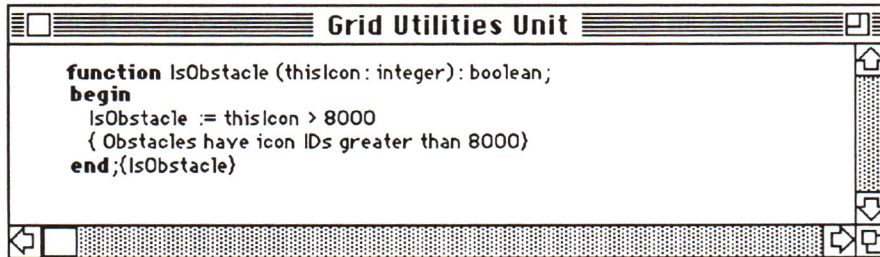


The IsCharacter function

Given an integer, which the function assigns to `thisIcon`, the function determines if that integer corresponds to a character icon, which must have a resource ID between 3000 and 5000. The function returns `true` if it does; otherwise, it returns `false`.

IsObstacle

This function is also defined in the Grid Utilities unit.

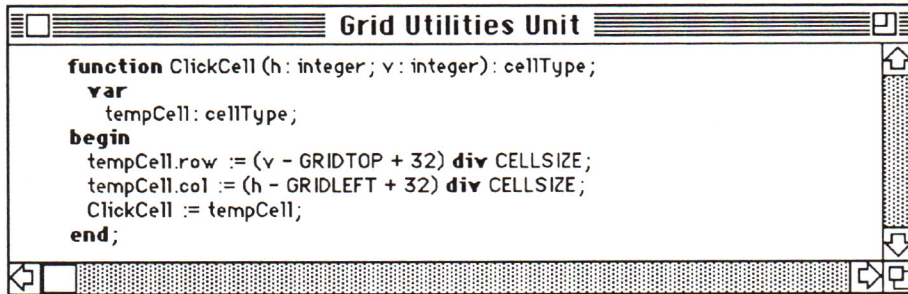


The IsObstacle function

This function returns `true` if the integer it is passed is greater than 8000. If so, it represents an obstacle, because obstacle icons have resource IDs greater than 8000. For example, the resource ID of the bomb is 8020.

ClickCell

The `ClickCell` function is defined in the Grid Utilities unit.



The ClickCell function

As you can see, the function takes two integers (points on the Macintosh screen) and returns a `cellType` variable as a result. The function calculates the row and column fields of the cell like this:

```

tempCell.row := (v - GRIDTOP + 32) div CELLSIZE;
tempCell.col := (h - GRIDLEFT + 32) div CELLSIZE;
    
```


Two integers, *h* and *v*, represent the coordinates of a point on the screen. The constants `GRIDTOP` and `GRIDLEFT` (defined in the `Globals` unit) mark the upper left corner of the grid. `CELLSIZE` is the size of each cell. The `div` operator behaves exactly like `/`, except that it returns an integer result.

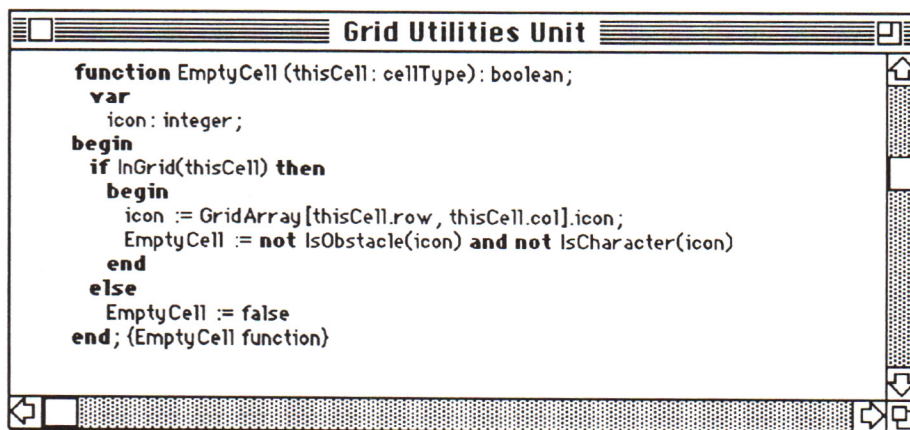
Since the cells in the grid are 32 pixels wide, we divide the horizontal position of the mouseclick *h* by `CELLSIZE`. But we have to subtract `GRIDLEFT` from *h* (because the grid is offset by that amount) and add 32 (because the first column is 32 pixels from the left).

`ClickCell` is called from the main event loop when the mouse is clicked to determine precisely which cell in the grid it was clicked in:

```
mouseDown :  
  begin  
    whereIsIt := FindWindow(thisEvent.where, thisWindow);  
    whichCell := ClickCell(thisEvent.where.h, thisEvent.where.v);
```

EmptyCell

The function `EmptyCell` in the `Grid Utilities` unit uses `IsCharacter` and `IsObstacle` to determine if a cell is empty.



The EmptyCell function

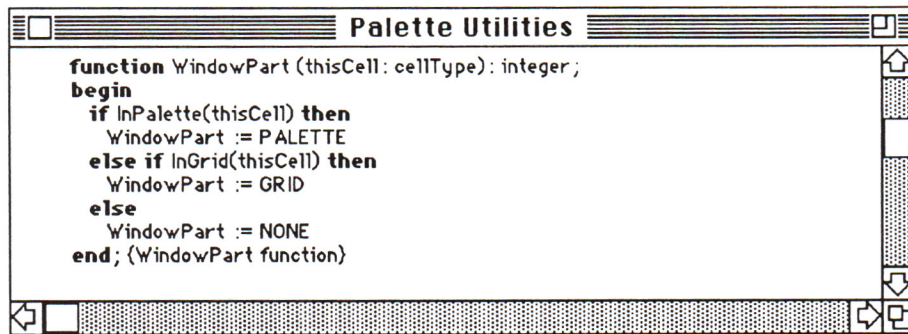
Any cell that contains neither an obstacle nor a character is assumed to be empty. You'll use this function at a later stage, when you start dealing with mouse clicks in the grid.

Activating the Palette

The function `WindowPart` and the procedure `SwitchPaletteIcon` in the `Palette Utilities` unit are useful for dealing with mouse clicks in the palette.

WindowPart

Once the program has determined which cell the user has clicked in, it must decide if that cell is in the grid or the palette. The function `WindowPart` in the `Palette Utilities` unit performs this task. It takes a cell as an argument and returns an integer corresponding to a part of the Drawing window. The global constants, `PALETTE` and `GRID`, define the two areas where the mouse click can occur.



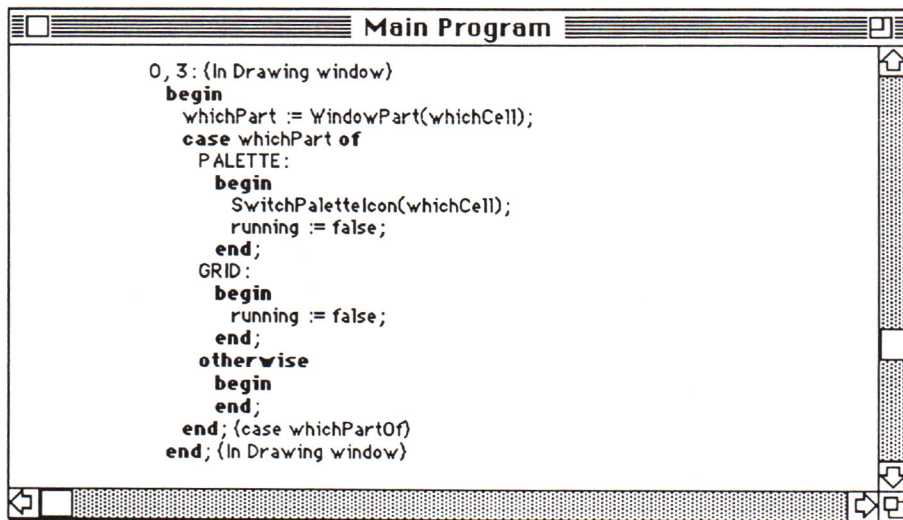
The function WindowPart

The constants `PALETTE`, `GRID`, and `NONE` are defined in the `Globals` unit.

Using WindowPart in the main event loop

The function `WindowPart` determines which part of the window the user has clicked—the palette, the grid, or neither. The main event loop uses the `WindowPart` function and a **case** statement to respond to the mouse click.

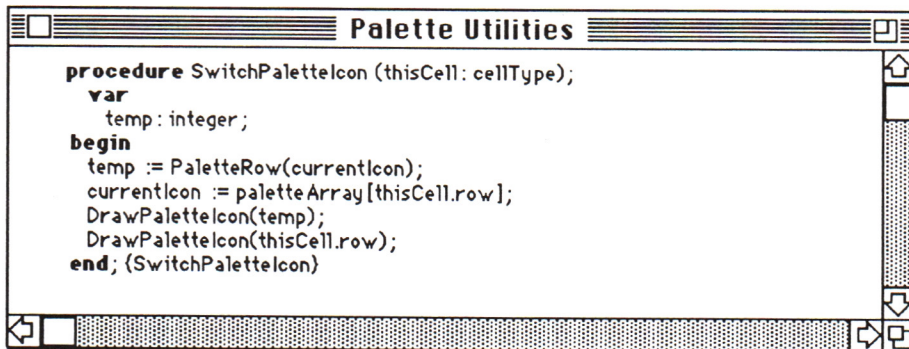
Different actions are taken for mouse clicks in the grid and mouse clicks in the palette. At this point, the actions are simple. If the mouse is clicked in the palette, the palette icon is inverted (using the procedure `SwitchPaletteIcon`) and the characters stop moving. If the mouse is clicked in the grid, the characters also stop moving. If the mouse is clicked in the menu bar, the characters *start* moving.



The case of a mouse click in the Drawing window

SwitchPalettelcon

Examine the `SwitchPaletteIcon` procedure in the Palette Utilities unit. This procedure does several things.



The procedure SwithcPalettelcon

- It stores the row of the global variable `currentIcon` in the local variable `temp`.
- It assigns to `currentIcon` the value of the icon the user has clicked on.
- Using the information in `temp`, it calls `DrawPaletteIcon` to uninvert (return to normal) the old `currentIcon` (taking advantage of the fact that `DrawPaletteIcon` inverts the icon only if it is the currently selected icon).
- It inverts the new `currentIcon`.

Summary of the Program Logic

The logic of the main event loop now goes something like this:

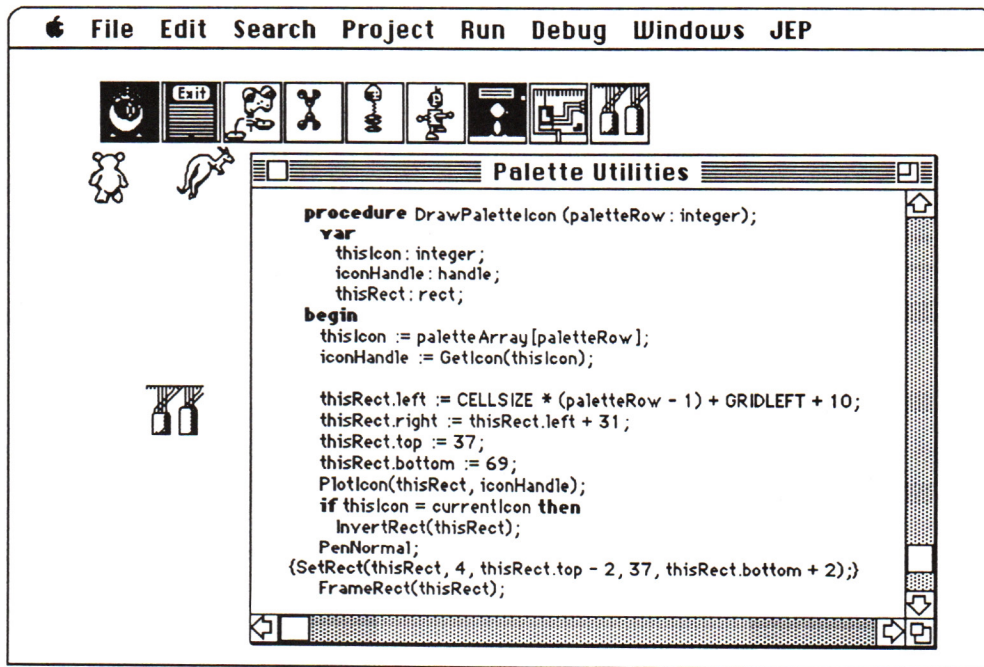
1. Has the user clicked the mouse?
2. If so, where?
3. If in the Drawing window, which part of the Drawing window?
4. If in the palette, then change the currently selected icon and stop the characters. If in the menu bar, start the characters. If in the grid, stop the characters.
5. Move the next character if `running` is `true`.
6. Repeat until `gameOver`.

Tinkering

The Tinkering section explores the main event loop and suggests various ways it can be changed.

Challenge

In `GridWalker`, the palette is drawn vertically down the left edge of the Drawing window. Can you draw it along the top of the screen instead?



GridWalker with the horizontal palette described in Tinkering

Stage Fifteen

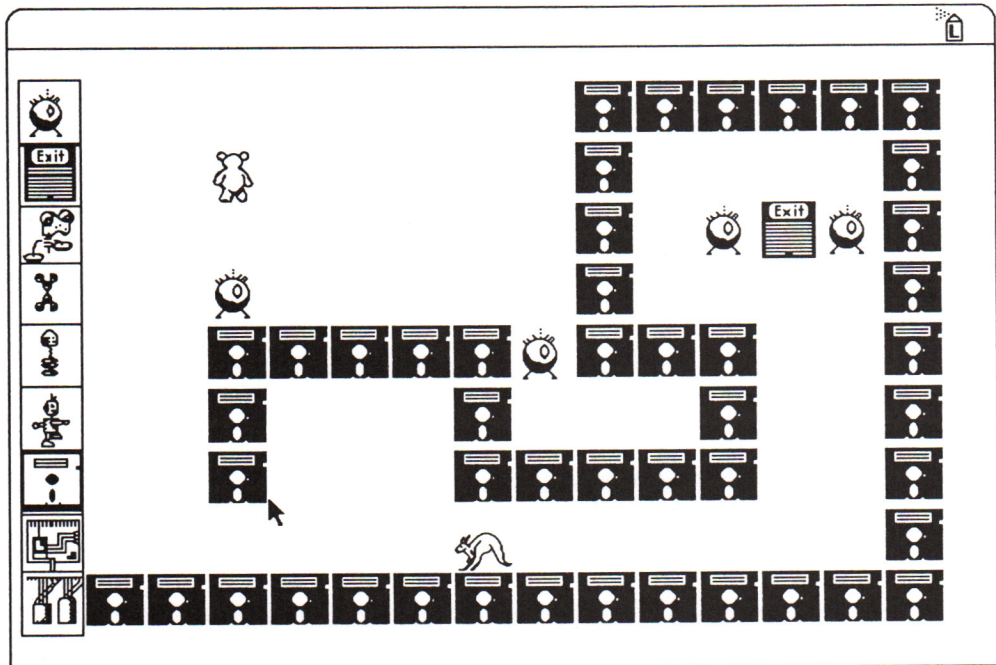
Activating the Grid

Problem

Up to this point the program displays the palette and lets the user select icons, but the palette still doesn't work completely. That is, the user can't do anything with the selected icons.

Solution: Adding Obstacles with the Mouse

In this stage, you'll make it possible for users to select obstacles from the palette and add them to the maze. Users will also be able to erase existing obstacles from the maze using the mouse.



The user positions obstacles in the maze

Assembly

In Stage Fifteen you:

- add two procedures, `MouseDown` and `GridClick`, to the Array Management unit
- add calls to those procedures in the main program

Explanation The Procedure `MouseDown`

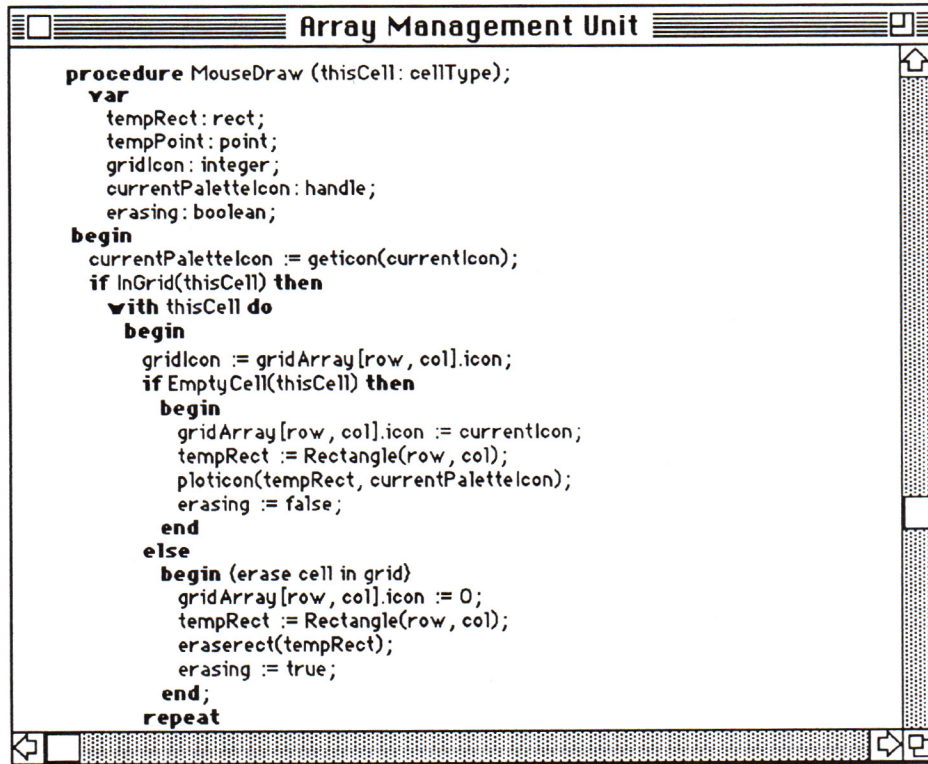
The `MouseDown` procedure in the Array Management unit is used to draw and erase obstacles. At this point, it does not create characters.

When the user clicks the mouse, `MouseDown` first determines if the pointer is in the grid; if it isn't, the click is ignored. If the pointer is in the grid, the procedure checks to see if that cell in the grid is empty. If it is, `MouseDown` draws the currently selected palette icon, updates `gridArray`, and sets the boolean variable `erasing` to `false` (the icon is not being erased).

```
if EmptyCell(thisCell) then
  begin
    gridArray[row, col].icon := currentIcon;
    tempRect := Rectangle(row, col);
    PlotIcon(tempRect, currentPaletteIcon);
    erasing := false;
  end;
```

If the cell in the grid is *not* empty, the obstacle in that cell is erased, `gridArray` is updated, the `QuickDraw` procedure `EraseRect` is called, and `erasing` is set to `true`.

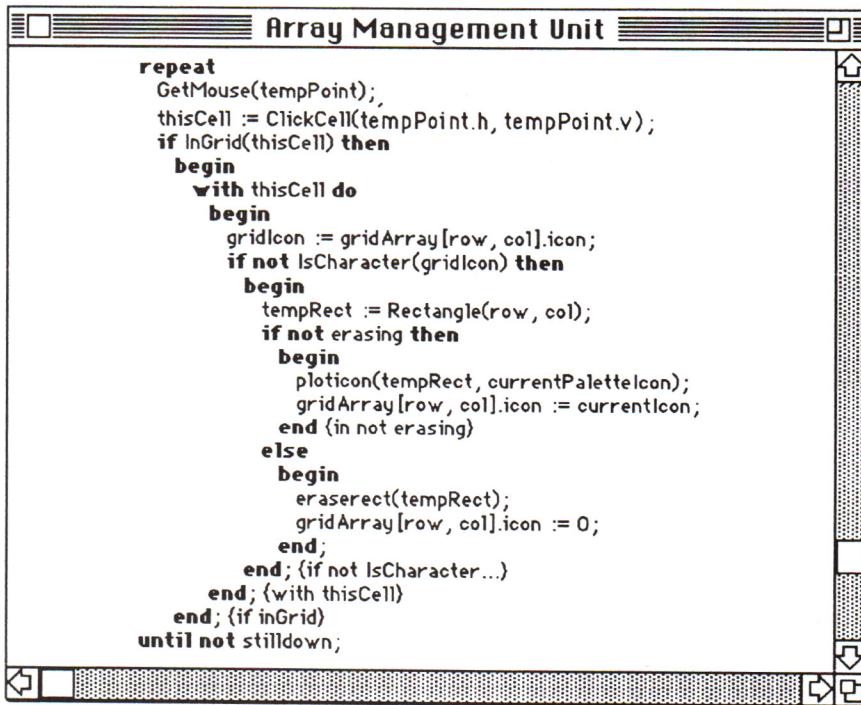
```
else
  begin
    {erase cell in grid}
    gridArray[row, col].icon := 0;
    tempRect := Rectangle(row, col);
    EraseRect(tempRect);
    erasing := true;
  end;
```



The beginning of the MouseDraw procedure

The **repeat** loop at the end of MouseDraw executes until the mouse button is released and the Toolbox global variable `stilldown` is therefore set to `false`. This loop continues the drawing process that was initiated at the beginning of the procedure.

For example, if the initial mouse click is on an empty cell and an obstacle is drawn in that cell, that obstacle continues to be drawn as the mouse is dragged across other cells, until the mouse button is released. This allows the user to “paint” across the grid with icons. If the initial mouse click erases an obstacle, this erasing continues until the mouse button is released. In both cases, any mouse clicks outside the grid are ignored.



The repeat loop in MouseDraw

The **repeat** loop identifies the position of the mouse (contained in a record called tempPoint) with the Toolbox procedure GetMouse:

```

GetMouse (tempPoint);
thisCell := ClickCell (tempPoint.h, tempPoint.v);

```

See *Inside Macintosh*, I-259, for more information about GetMouse.

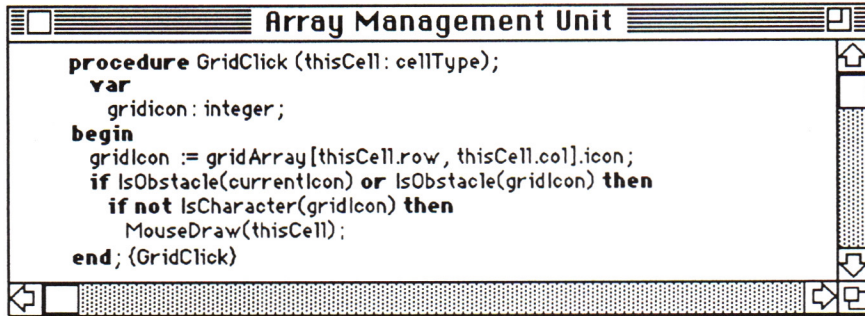
The GridClick Procedure

The JEP procedure GridClick, added to the Array Management unit, is called when the main event loop detects a click anywhere in the grid.

```

GRID:
  begin
    running := false;
    GridClick(whichCell);
  end;

```



The GridClick procedure

First, the procedure looks in `gridArray` to determine which icon (if any) is in the cell the user clicked. It assigns the resulting integer to the local variable `gridIcon`.

```
gridIcon := gridArray[thisCell.row, thisCell.col].icon;
```

Then it executes the following logical statement.

```

if IsObstacle(currentIcon) or IsObstacle(gridIcon) then
  if not IsCharacter(gridIcon) then
    MouseDraw(thisCell);

```

This statement uses nested `if` statements. The conditional expression of the first `if` statement is evaluated; if it is `true`, then the conditional expression of the second `if` statement is evaluated. If that expression is `true`, then `MouseDown` is executed. If either condition is `false`, `MouseDown` is not executed.

JEP Note

Because users can now use the mouse to draw their own mazes, you no longer need the `RandomMaze` procedure in the main program.

Tinkering

Sounds

The Tinkering section in this stage explores the different kinds of sounds that can be produced on the Macintosh.

Challenge

Have the program generate the sound of an explosion (KABOOM) whenever the user clicks in a palette cell containing the bomb icon.

Stage Sixteen

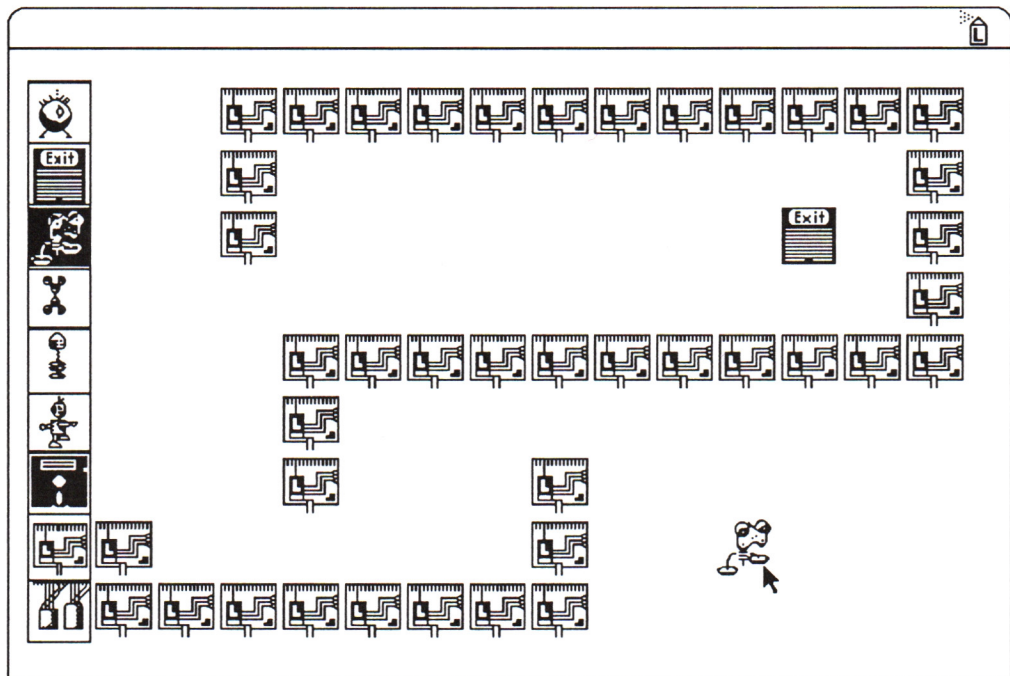
Adding and Deleting Records

Problem

Now users can create their own mazes by selecting obstacle icons from the palette and then drawing with the mouse in the grid. But only you, the programmer, can put characters in the maze.

Solution: Adding and Deleting “Character” Records

In this stage, you add routines that give the user the ability to add and delete characters.



The user clicks characters into the grid

Assembly

In Stage Sixteen you add:

- the procedures `CreateCharacter` and `DeleteCharacter`
- statements to the `GridClick` procedure

Explanation Creating New Characters

At the end of this stage, the user can add character records to `characterArray` by selecting one of the character icons in the palette and then clicking anywhere in the grid. The character is inserted in the cell the pointer was in when the mouse is clicked. In this way new characters are added by the user rather than the programmer.

JEP Note

You've created a palette with four character icons—JEP, TUMBLER, BOUNCER, and ROBOT. You can change these icons by editing the `InitPalette` procedure in the `Palette Utilities` unit (see Stage Twelve). Replace the constants for characters with others that are available (see page 174 for a list). You can also change the obstacles in the palette.

currentStrategy

The variable `currentStrategy` (of type `strategyType`) is added to the `Globals` unit. When a character is created, its strategy is set to `currentStrategy`.

At this stage in the development of `GridWalker`, `currentStrategy` is set to `randomWalker` at the top of the main program. In a later stage, you'll let the user change the current strategy with a menu selection.

totalCharacters

The global variable `totalCharacters` is set to 0 when the program starts. This value changes as new characters are created and deleted.

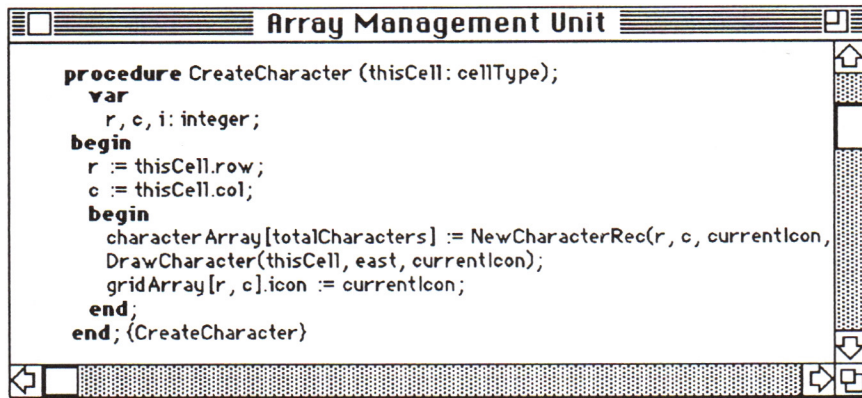
The lines that originally set two characters into the grid at the beginning of the program are no longer needed and are erased in this stage.

JEP Note

Variables like `currentIcon`, `currentStrategy`, and `totalCharacters` might be called **project globals**. Because they are declared in the Globals unit, which is first in the build order and is used by all other units, they are available to all functions and procedures in the project including the main program.

CreateCharacter

The procedure `CreateCharacter` is defined in the Array Management unit.

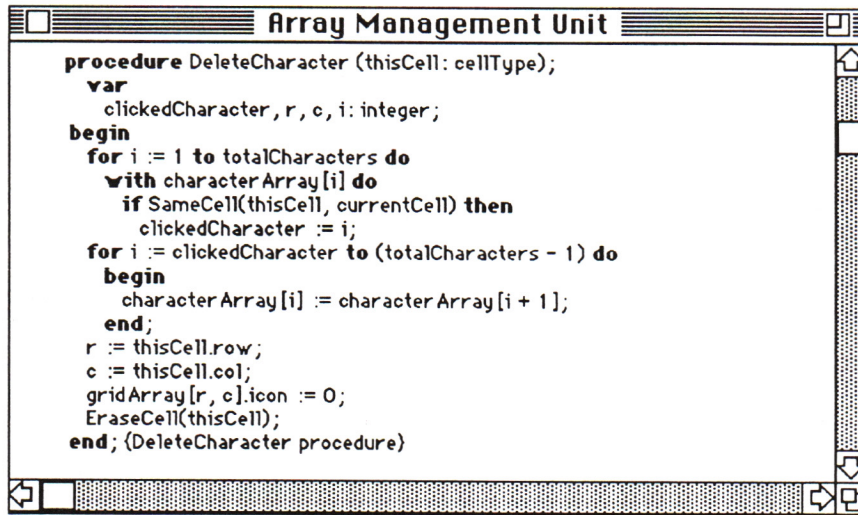


The CreateCharacter procedure

- This procedure calls the `NewCharacterRec` function to add a new record to `characterArray`. Since you've already incremented `totalCharacters` (in `GridClick`), `totalCharacters` now indexes the new final element in the array.
- It draws the new character in the specified cell, facing east.
- It makes changes to the `icon` field of `gridArray` to indicate that there is now an icon corresponding to the new character's row and column position.

DeleteCharacter

The procedure `DeleteCharacter` is also defined in the Array Management unit.



The DeleteCharacter procedure

This procedure performs the following tasks:

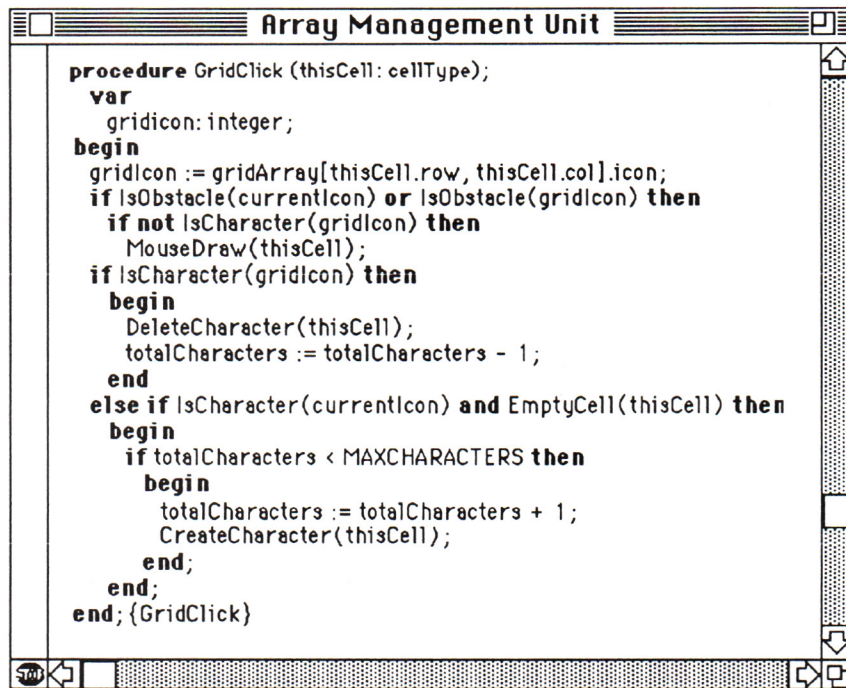
- erases the character's record from the character array (by writing over it)
- updates `gridArray`
- erases the appropriate cell

Notice how the array is compressed by copying up one record at a time. This is a standard way of deleting records from an array cleanly:

```
for i := clickedCharacter to (totalCharacters - 1) do
begin
    characterArray[i] := characterArray[i + 1];
end;
```

GridClick

You also add some statements to the `GridClick` procedure in the Array Management unit that allow characters, as well as obstacles, to be added to the grid.



The GridClick procedure

If the current palette icon is an obstacle or if the user has clicked on an obstacle in the grid (but not on a character), then MouseDraw is called. If the user *has* clicked on a character in the grid, that character is deleted with these statements:

```

if IsCharacter(gridIcon) then
    begin
        DeleteCharacter(thisCell);
        totalCharacters := totalCharacters - 1;
    end;

```

If the user has not clicked on a character the following **else if** condition is executed. If the current palette icon represents a character, and the user has clicked on an empty cell in the grid, and totalCharacters is less than MAXCHARACTERS, then a new character is created and totalCharacters is incremented:

Just Enough Pascal

```
else if IsCharacter(currentIcon) and
      EmptyCell(thisCell) then
  begin
    if totalCharacters < MAXCHARACTERS then
      begin
        totalCharacters := totalCharacters + 1;
        CreateCharacter(thisCell);
      end;
  end;
```

JEP Note

When a character is created, the character's icon will be the current palette icon (currentIcon) and the character's strategy will be currentStrategy.

Tinkering

The Tinkering section in this stage suggests additional ways to create characters in the grid.

Challenge

As the program is now written, there is no relationship between a character's icon and its strategy. For example, you can have a bear walking randomly or a bear skirting the grid. Can you figure out a way to link icon and strategy, such that bears always walk randomly (for example) and kangaroos always skirt the grid?

Using the Toolbox procedure SysBeep(10), add a statement that makes the beep sound if the user tries to add characters when totalCharacters = MAXCHARACTERS.

Stage Seventeen

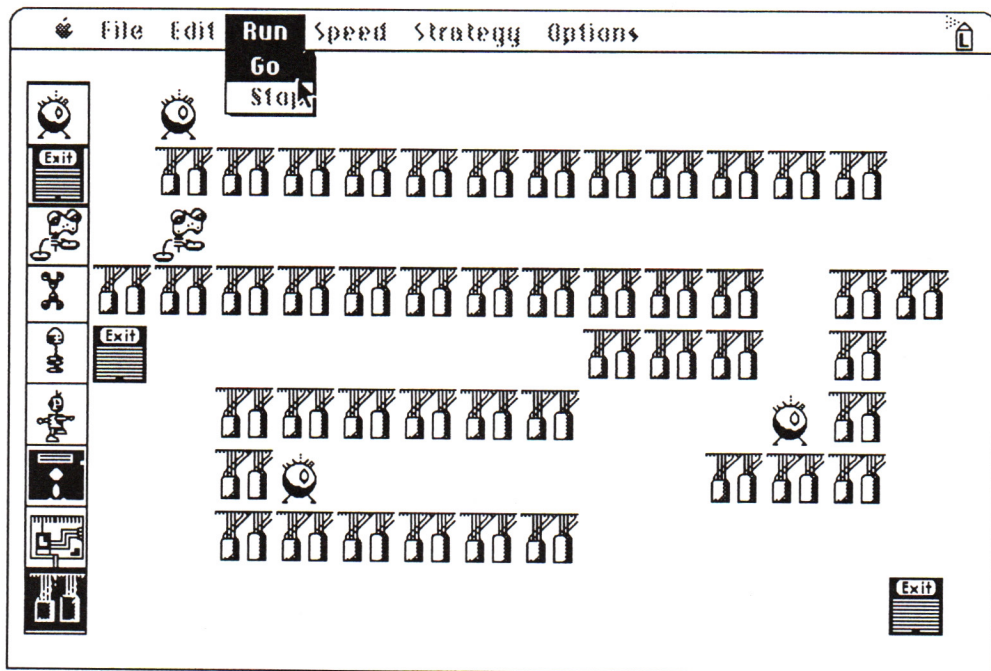
The Menu Bar

Problem

Although users can now click characters and obstacles into the grid from the palette, other functions, such as assigning maze-solving strategies and setting animation speed, are still handled by the programmer.

Solution: The Menu Bar

In this stage you will add a standard Macintosh menu bar to give users additional control of the program. At the end of this stage, the only active menu will be the Run menu.



The project with the menu bar

Assembly

In Stage Seventeen you add:

- the Menu Utilities unit
- calls to the menu procedures `MakeMenus`, `DrawMenuBar`, and `InMenuBar`

Explanation Setting Up the Menu Bar

The seven GridWalker menus are defined in the resource file `JEP Resources.rsrc` along with the icons and sound resources. Like all the resources in `JEP Resources.rsrc`, the menu resources were created with the application `ResEdit`. You can use `ResEdit` to examine these resources and to alter them if you wish.

Two menus, the File menu and the Edit menu, are fully defined in that resource file. Both the menu names and item names are listed there. For the other menus, the resource file only lists the menu names. The program creates the item names for these menus with `Toolbox` procedure calls.

Menu Handles

Examine this line in the `Globals` unit:

```
appleMenu, ..., optionsMenu: menuHandle;
```

It sets up seven variables of the type `menuHandle` that will be used as references to the seven GridWalker menus.

Toolbox Note:

The attributes of each menu, the menu name and the items on it, are stored in a location in memory. This information is assigned a **handle**, which helps keep track of where the information is stored. A detailed description of handles, pointers, and other features of memory allocation can be found in *Inside Macintosh*.

MakeMenus

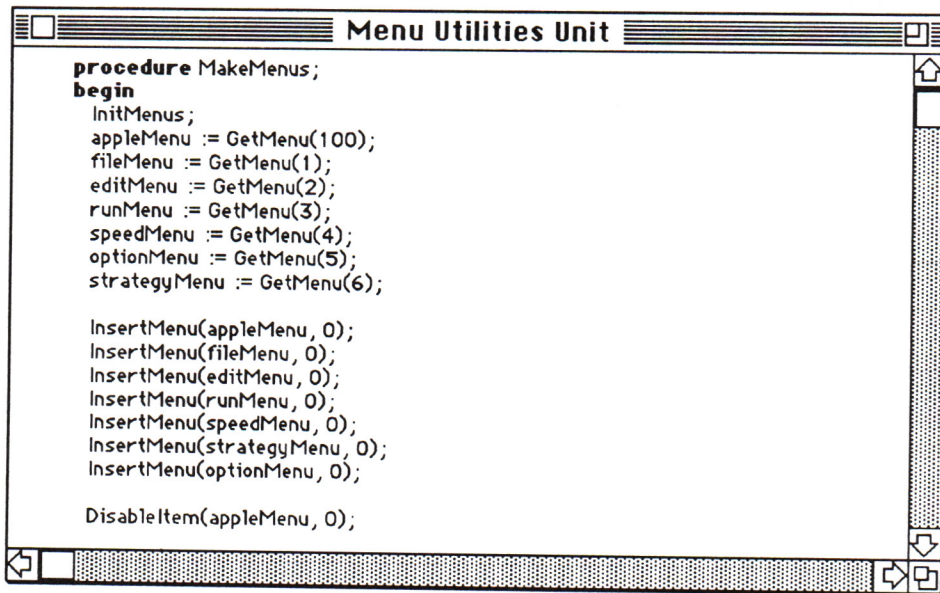
The JEP procedure `MakeMenus` is defined in the `MenuUtilities` unit. It calls several `Toolbox` routines.

GetMenu

Information about menus and the items on them is obtained from the resource file with the `GetMenu` procedure. For example, the following statement finds the information in the menu resource ID 1 and assigns that information to the handle `fileMenu`.

```
fileMenu := GetMenu(1);
```

In JEP Resources.rsrc the menu resource ID 1 defines the name of the menu as "File Menu" and includes a single item, "Quit."



The beginning of the MakeMenus procedure

InsertMenu

Next, these menus must be positioned on a menu bar. This is accomplished with the `InsertMenu` procedure.

```

InsertMenu(appleMenu, 0);
InsertMenu(fileMenu, 0);
InsertMenu(editMenu, 0);
InsertMenu(runMenu, 0);
  
```

Here four menus are added to the menu bar. The first argument indicates which menu is to be added. The second argument, an integer, indicates its position. When this integer is 0, the menu is added at the end of the list.

AppendMenu

As defined in the resource file, the Run menu has no items. These lines add the items **Go** and **Stop**:

```
AppendMenu (runMenu, 'Go');  
AppendMenu (runMenu, 'Stop');
```

Notice that the first argument in the `InsertMenu` `AppendMenu` procedure is a menu handle. The second argument is a text **string**—a list of consecutive characters (numbers and letters). The single quotes (') mark the characters as a text string. In this case, the text string becomes the menu item.

DisableItem

`DisableItem` makes a menu item appear in gray, signifying that it cannot be selected. If 0 is given as the argument, the entire menu is disabled.

JEP Note

When you first add the Menu Utilities unit to the project, the `MakeMenus` procedure disables all the menus. To enable a menu, delete the corresponding call to `DisableItem`.

EnableItem

`EnableItem` is a Toolbox procedure that enables a previously disabled menu item. Once the item is enabled, the user can select it from the menu.

Using the MenuSelect Function

The Toolbox function `MenuSelect` is used to determine which menu has been selected. The `InMenuBar` procedure, declared in the Menu Utilities unit, uses this function. It calls the function with the statement

```
thisone := MenuSelect (myevent.where);
```

`MenuSelect` takes a point as an argument (the point is the position of the cursor when the mouse was clicked) and returns a variable `thisOne` of type `longint`. `MenuSelect` also automatically pulls down menus and highlights items as the mouse is dragged along them.

longint

The type `longint`, a “long” integer, holds twice as much information as a regular integer. Each `longint` variable can be broken apart into two regular integers using the `ToolBox` functions `HiWord` and `LoWord`.

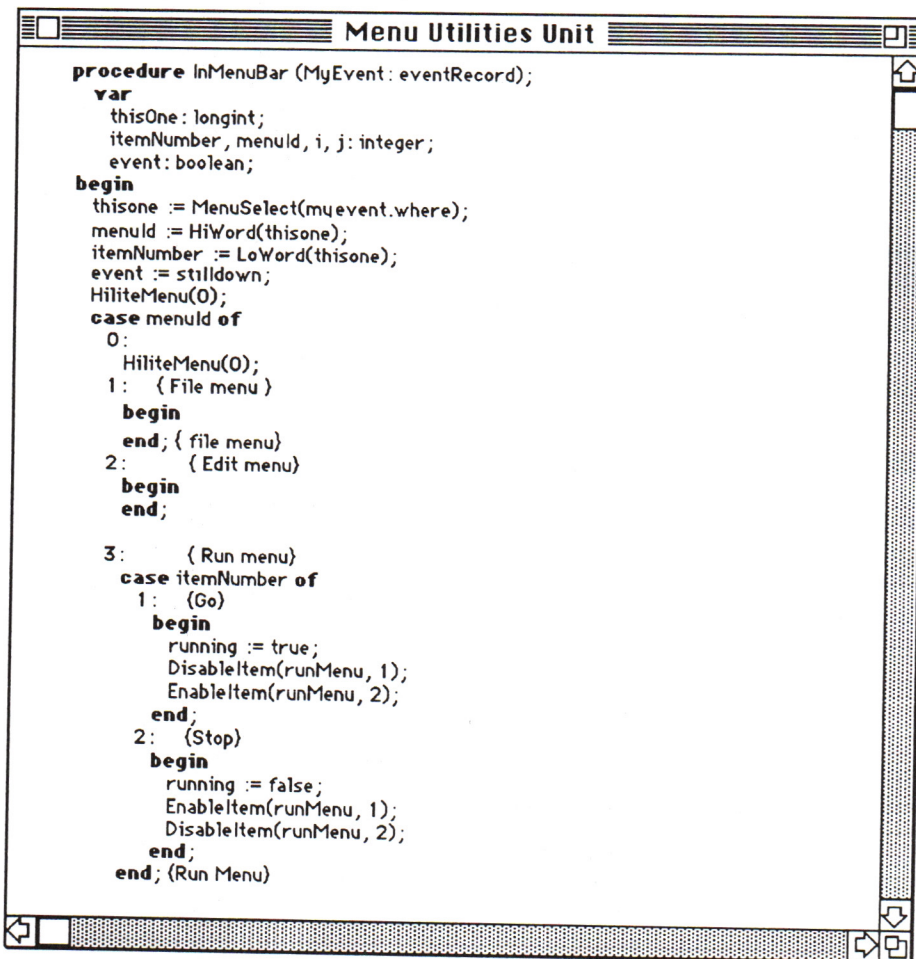
thisOne

The variable `thisOne` is a variable of the type `longint`. It has two components that record the menu ID of the menu (the **high order word**) and the item number on that menu (the **low order word**).

These two statements assign the two components of `thisOne` to two separate integers:

```
menuId := HiWord(thisOne);
itemNumber := LoWord(thisOne);
```

Using two nested **case** statements, the `InMenuBar` procedure now takes appropriate action based on the integer values of `menuId` and `itemNumber`.



The InMenuBar procedure

For example, if `menuID = 3` (the Run menu) and `itemNumber = 1` (the **Go** item), then the procedure does three things:

- It sets `running` to `true`.
- It disables the **Go** item.
- It enables the **Stop** item.

Later we will add other cases to the `InMenuBar` procedure.

Disabling Menu Items on the Fly

If there are no characters in the grid, the **Go** and **Stop** items on the Run menu must be disabled. This is accomplished with the procedure `ResetRunMenu`, in the Menu Utilities unit, which examines the value of the variables `totalCharacters` and `running` and sets the Run menu correctly. For example, if there are no characters in the grid (`totalCharacters = 0`), then both the **Go** and **Stop** items are disabled.

You add a call to this procedure to the main program, at the end of the **case** statement.

Tinkering

The Tinkering section of this stage outlines the steps you should follow to add an item to a menu. As an example, it describes the process of adding **Pause** to the Run menu. When you select **Pause**, the animation pauses for 5 seconds and then continues normally.

Challenge

Add a **Reset** item to the Run menu. When the user chooses **Reset**, all characters return to their starting positions. To make this work, in addition to adding the item to the menu, you'll have to add a field to `characterRecType` that records the starting position for each character. You'll also have to erase the character icons from the screen and update `gridArray`.

Stage Eighteen

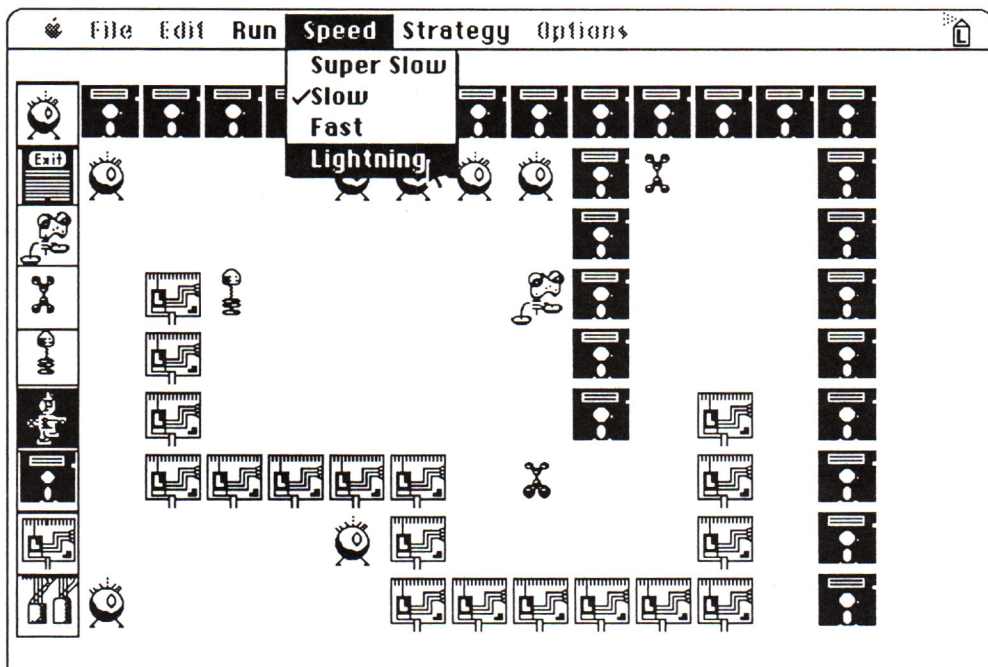
Adding Menus

Problem

The menu bar is now in place, but users still cannot change the speed of the animation or set strategies for individual characters.

Solution: Adding Menus

In this stage you activate the Speed and Strategy menus.



Using the Speed menu

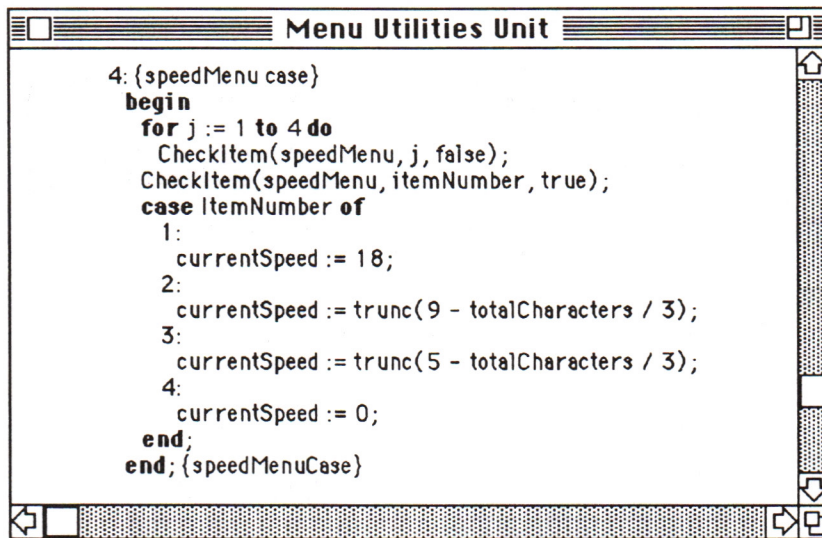
Assembly

In Stage Eighteen you:

- add statements to the JEP procedure `InMenuBar` to handle new cases
- append items to the Speed Menu
- activate the Speed and Strategy menus

Explanation Checking and Unchecking Menu Items

Statements added to the `InMenuBar` procedure allow the user to change `currentSpeed` with a menu selection.



Changes to the `InMenuBar` procedure

When the Speed menu is selected, the **for** loop first unchecks all four items. Then it places a check next to the selected item. Finally, the **case** statement sets `currentSpeed`.

As the variable `currentSpeed` (the length of the delay interval) decreases, the speed of the animation increases. When the last item is selected (item 4, **Lightning**), `currentSpeed` is set to 0, meaning there is no delay interval at all.

Activating the Strategy Menu

When you add a strategy to the project, you must do three things to make it appear on the menu.

- Make the constant NUMSTRATEGIES (in Globals) equivalent to the total number of strategies you have defined (e.g. if you have four strategies, then NUMSTRATEGIES = 4).
- Append the item to the Strategy menu in the MakeMenus procedure.
- Add a case to the InMenuBar procedure.

JEP Note:

When you add a new strategy to the project, you'll have to develop the logic of the strategy in its own procedure, add its name to the list of strategies defined in `strategyType`, and add a new case to the `StrategyHeading` function. You'll actually do this in the next assembly stage.

The AppendMenu Procedure

Calls to the Macintosh Toolbox procedure `AppendMenu` append new items to the designated menus. For example, the following statements add the items `GridSkirter` and `RandomWalker` to the Strategy menu.

```
AppendMenu(strategyMenu, 'GridSkirter');
AppendMenu(strategyMenu, 'RandomWalker');
```

Other lines add items to the Speed menu. When you add your own strategies, you must include a call to `AppendMenu` with the new strategy name as an argument.

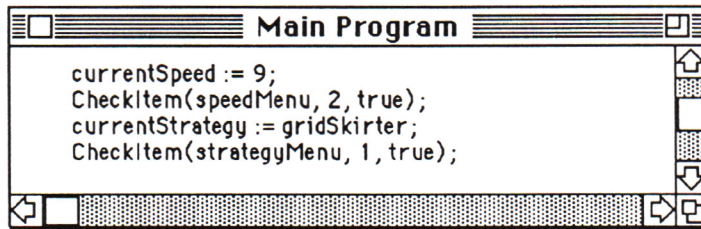
Selecting a strategy from the menu

The current strategy is the checked item on the Strategy menu. When an item is selected from the Strategy menu, certain statements in the `InMenuBar` procedure are executed.

First, all items on the menu are unchecked. Then the selected item is checked. Finally, the **case** statement sets the variable `currentStrategy` appropriately. If you add a strategy, add an additional case.

Changes to the Main Program

The following statements are added to the beginning of the main program to set the starting values for `currentSpeed` and `currentStrategy` and to place check marks next to the appropriate items on the menus.



Changes to the main program

The Toolbox procedure `CheckItem` takes three arguments: the handle for a menu, the item number, and `true` (to check the item) or `false` (to uncheck it).

Tinkering

The Tinkering section in this stage describes the steps you should follow to add a menu to the menu bar. As an example, the Instructions describe the process of adding a “Volume” menu to the menu bar. You can change the volume of the various sounds by selecting from this menu.

Challenge

The JEP obstacle icons fall into two groups—high-tech (FANPLUG, CIRCUIT, FLOPPYDISK, etc.) and low-tech (ROCK, HEDGE, FENCE, etc.). Add additional menu item that let the user choose between these two sets. When Low-Tech is selected, for example, the palette is drawn with rocks, hedges, and fences.

Stage Nineteen

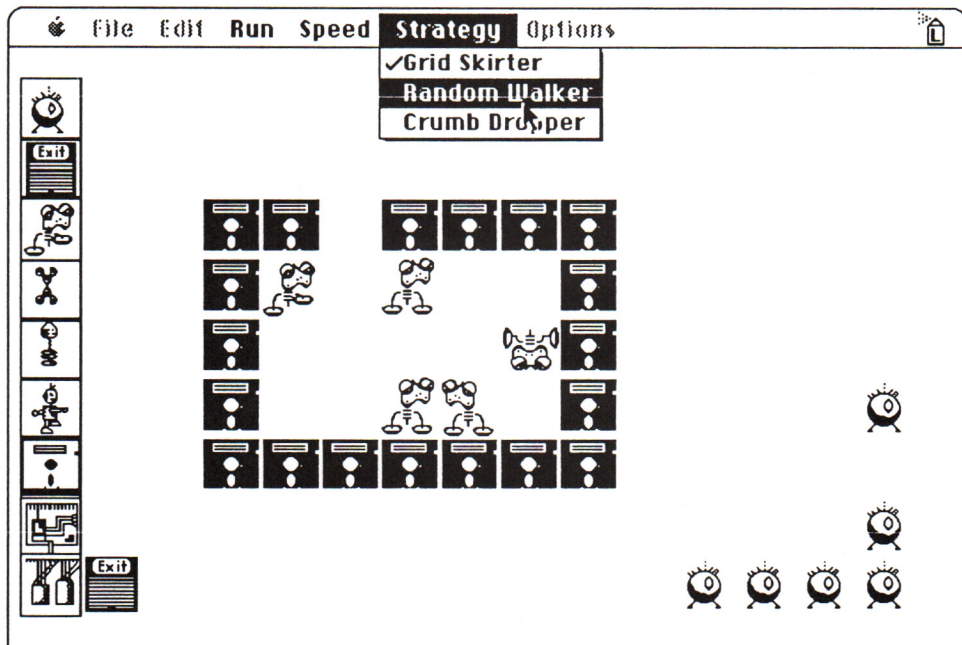
Adding a Strategy

Problem

The two strategies that you have to work with, GridSkirter and RandomWalker, suffer some obvious shortcomings. You've probably noticed, for example, that a character using GridSkirter frequently gets stuck in a fruitless circling pattern, recovering only when another character comes along and forces it to move in a new direction. Although a character using RandomWalker will eventually find an exit (if it doesn't run into a bomb first), it lacks any sense of purpose.

Solution: CrumbDropper

In this stage, you implement a third strategy, called CrumbDropper. Characters using this strategy do significantly better than GridSkirters or RandomWalkers because they have a way of recording where they've been.



Selecting from the Strategy menu

Assembly

In Stage Nineteen you add:

- a field to `gridCellType` to record the number of “crumbs” in a cell
- procedures to the Strategies unit
- a change to `InitGrid` in the Array Management unit
- changes to the Menu Utilities unit

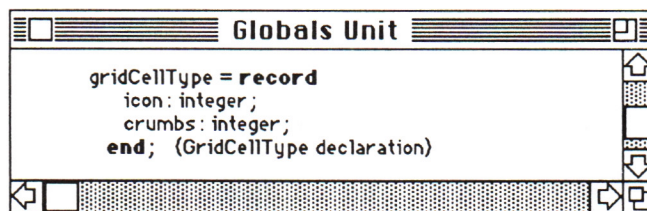
Explanation Creating a New Strategy

The new strategy `CrumbDropper` works like this: When a `CrumbDropper` character enters a cell, it drops a “crumb” in that cell. Although these crumbs do not appear on the screen, the program records how many have been dropped in each cell. When the character decides on its next move, it selects the legal cell that contains the fewest crumbs.

The `CrumbDropper` involves changes in several places in the project. You add a new field in the `gridCellType` to record the crumbs that characters drop and a procedure that drops crumbs into a cell when the character moves there. You also add a new function `LeastCrumbsHeading` that identifies a heading to an adjoining legal cell with the least number of crumbs. Finally, you add a new case to the `StrategyHeading` function.

Crumb Field

The new `crumb` field that you add to the declaration of `gridCellType` in the `Globals` unit is used for keeping track of how many crumbs have been dropped in a cell.



New declaration of `gridCellType`

- The line you add to `InitGrid` sets the starting number of crumbs in each cell to 0.
- The `DropCrumb` procedure “drops” a crumb in a particular cell. That is, when a character using the `Crumb Dropper` strategy enters a cell, `DropCrumb` increases the value in the `crumb` field (of `gridArray`) for that cell by 1.
- The `NumCrumbs` function reports how many crumbs there are in a specified cell.

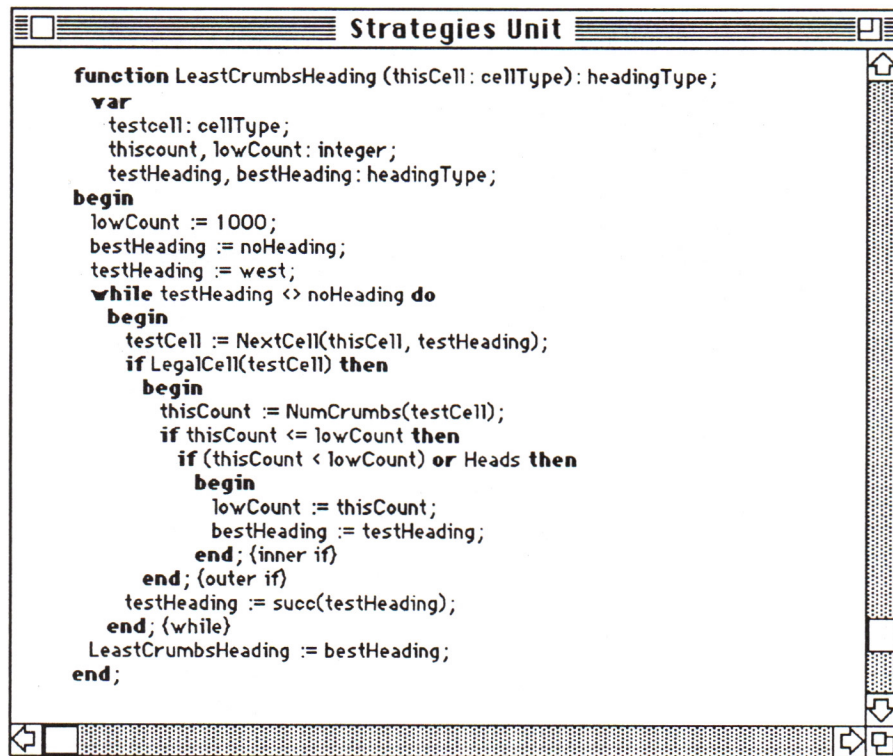
The While Loop

The function `LeastCrumbsHeading` makes use of another iterative technique called the **while** loop. Like other Pascal loops, the **while** loop has a test condition and a program block. The test condition (in this case the boolean expression `testHeading <> noHeading`) is evaluated. If it is true, the block is executed and the test condition is evaluated again. This continues until the test condition is not true. The program executes the statement that follows the block.

With a **repeat** loop, the test condition is at the end of the loop; it will always execute at least once. Because the test is at the beginning of a **while** loop, sometimes it will not execute at all. In `LeastCrumbsHeading`, because `testHeading` is initially set to `noHeading`, the loop is always executed at least once.

The LeastCrumbsHeading function

The `LeastCrumbsHeading` function determines which of the four cells adjoining a given cell contains the least crumbs.



The LeastCrumbsHeading function

How LeastCrumbsHeading works

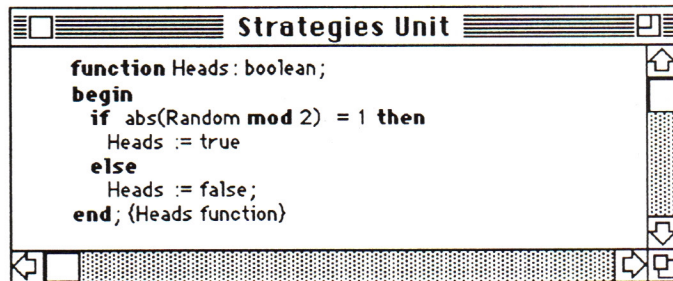
- First, the function sets `lowCount` to 1000 (an arbitrarily large number) and `bestHeading` to `noHeading`. The function now tries to find a heading that has fewer crumbs than `lowCount`.
- The function tests the adjoining cell in each of the four possible headings: west, north, east, and south.
- If the number of crumbs in the tested cell is less than the current minimum (`lowCount`), or if the number of crumbs in the tested cell is the same as `lowCount` and the function `Heads` (which returns randomly `true` or `false`) returns `true`, `bestHeading` takes on the value of the tested heading.
- At the conclusion of the four tests, the result of the function `LeastCrumbsHeading` is set to the current `bestHeading`.

JEP Note

The use of the coin flip ensures that a character using the Crumb Dropper strategy won't always head in the same direction in cases where the number of crumbs in each cell is the same.

The Heads function

`Heads` is a JEP function defined in the `Strategy` unit and used in the `LeastCrumbsHeading` function. It simulates a coin flip. It returns `true` approximately half the time, and `false` half the time.



The Heads function

This expression divides the absolute value of the number returned by the `QuickDraw` function `Random` by two and returns the remainder:

```
abs(Random mod 2)
```

The result is either 0 (for an even number) or 1 (for an odd number). Review the discussion in Stage Eight if you need a refresher on random numbers.

The DropCrumb and EatCrumb Procedures

When a character using the CrumbDropper strategy enters a cell, the crumbs field for that cell is incremented by the procedure DropCrumb in the Strategies unit.

JEP Note

The procedure EatCrumb is provided in this stage for you to experiment with. This procedure subtracts 1 from the integer stored in the crumbs field of an element of gridArray.

StrategyHeading

The crumbDropper case is added to StrategyHeading to do two things:

- It “drops a crumb” in the current cell.
- It calls LeastCrumbsHeading to determine which adjoining cell has the least number of crumbs, and is therefore the preferred heading.

```

case strategy of
  gridSkirter:
    .
    .
    .
  crumbDropper:
    begin
      DropCrumb (cell);
      StrategyHeading :=LeastCrumbsHeading (cell);
    end; {crumbDropper case}

```

Changes to the Strategy Menu

In order to handle the new strategy, you change the menu bar. You add a case to the InMenuBar procedure and append an item to the Strategy menu.

Tinkering

The Tinkering section of this stage outlines the steps you should follow to add a strategy to the application. As an example, the Instructions describe the process of creating the strategy “ChalkWalker” and adding it to the Strategies menu.

Challenge

Create your own strategy and add it to the menu.

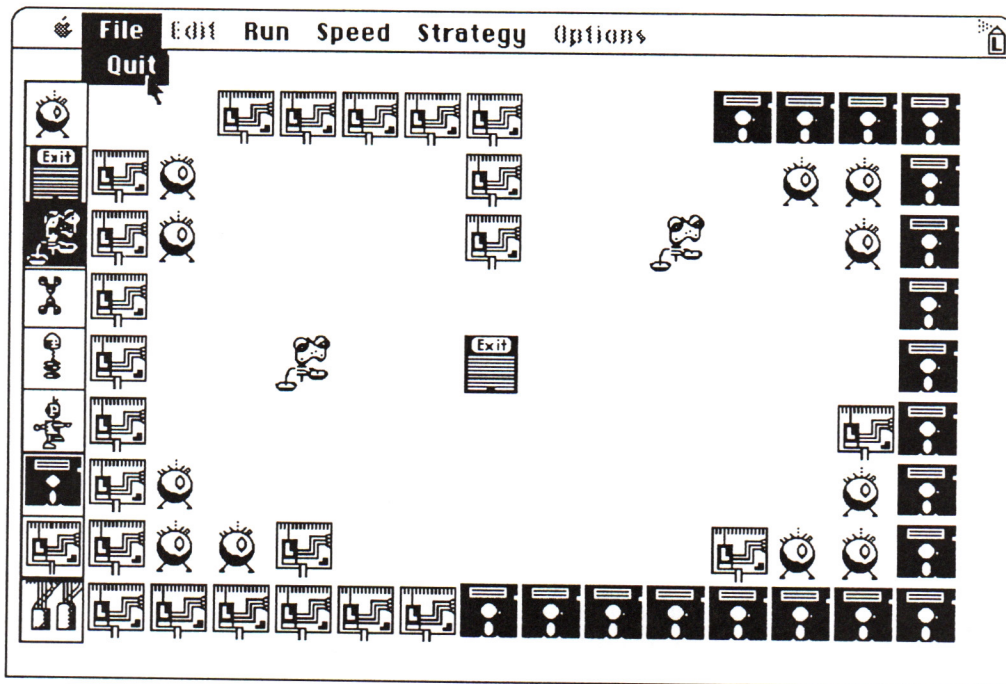
Stage Twenty Finishing Touches

Problem

The program now comes to a halt (and returns control to THINK Pascal) when any one of the characters hits a bomb or leaves through the exit. A more complete program would have the action continue until the user wants the program to halt. If one character encounters a bomb, so what? That just makes life in the maze a little easier for any remaining characters.

Solution: Quit

In this final stage you add **Quit** to the File menu, allowing the user to stop the program. The game continues until the user decides to quit the application by choosing Quit. In this stage you also build a stand-alone application that users launch from the Finder.



*Choosing **Quit** from the File menu*

Assembly

In Stage Twenty you:

- add procedures to the Animation unit
- enable **Quit** on the File menu
- build your project as a stand-alone application

Explanation Setting Flags

You add a variable `collision` to the main program that will be used as a flag. When a character runs into a bomb or an exit, this variable is changed from `false` to `true`. Later, when you animate the characters, this flag is used to handle a special case.

MoveCharacter

You change the `MoveCharacter` procedure so that when a character runs into a bomb or exit, that character is deleted from the array, `collision` is set to `true` and the program continues to run.

Condition

Finally, you add this condition to the main program:

```
if (totalCharacters > 0) and not collision then
    characterCount := (characterCount + 1) mod totalCharacters;
```

When a character collides with a bomb or exit, the counter variable `characterCount` is not incremented. The reason for this condition is simple. When a character is erased from the array, the next character in order takes its place. Without this condition, whenever a character was deleted, the next character would not be animated.

Quit

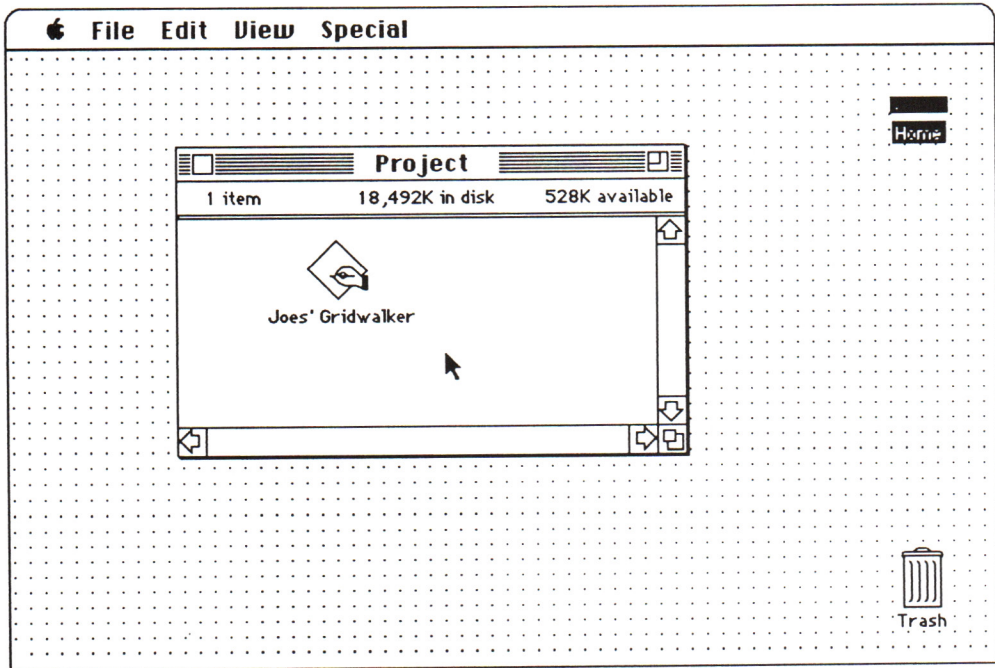
The `InMenuBar` procedure is changed to handle the case where **Quit** is selected from the File menu. In this case the variable `gameOver` is set to `true`. Now the main event loop can continue until the user chooses **Quit** from the File menu.

Build an Application

When you choose Build Application from the Project menu of THINK Pascal, you create a Macintosh application that can be executed independently from THINK Pascal.

- Users will be able to launch your application from the Finder by clicking its icon.

- The application contains only the executable (object) code for the program. Users will not be able to examine or change any of the source code or use any of the THINK Pascal debugging features.



The GridWalker application in the Finder

Finished?

You're finished only if you want to be. Here are some final challenges.

Individual Speed

In contrast to strategy, which can be different for each character, all characters in the maze now move at the same speed. What about making speed work like strategy, so that each newly created character is assigned the currently selected speed?

New Strategies

If you haven't done so already, you might try to design a strategy that is more efficient than CrumbDropper. For example, what about a strategy that always looks for the longest unobstructed path?

Vision

At it now stands, characters can “see” only one cell ahead (using the `NextCell` function). You might write an `OpenCellsAhead` function that would allow characters to look down a row or column, counting the number of open cells in a given direction. Building on this, you might also allow a character to “see” (and identify) another character at a distance.

Characters That Remember

Even if a character had access to a function that allowed it to look ahead, it wouldn't be much better off, because it still wouldn't be able to remember anything from one step to the next. Without memory, the characters cannot even devise and follow a simple plan, let alone learn from experience.

For example, suppose you want to allow a character to see an exit at a distance, then head in that direction. But suppose another character moves in the way, temporarily blocking the path. If the character doesn't have some way of remembering where it saw the exit, it might wander off again in the wrong direction.

Perhaps you should think about adding some additional “memory” fields to the character record.

Tag

You might also want to consider designing different games like tag. For example, you could add a new field to the character record called `it`. This would be a boolean variable that would keep track of whether a character was It or not. If a character that is currently It moves into a cell adjacent to a character that is not It (tags that character), then the characters switch roles. You'll probably have to make the character that gets tagged pause (skip a move) so that the other character can have a chance to get away.

Tag Plus Vision

If characters have the ability to see each other at distances, then you might work out separate strategies for characters that are It (move in the direction of characters that are not It) and not It (move away from characters that are It).

Other Games

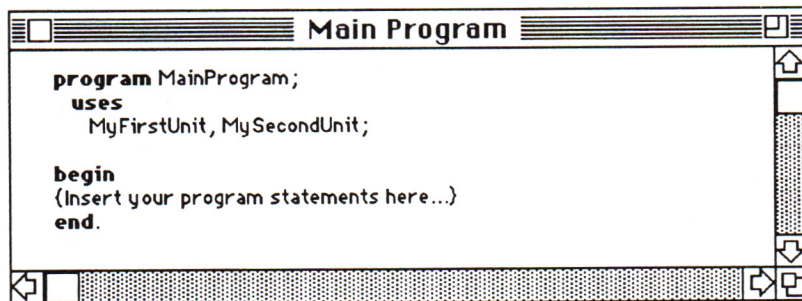
Other games you might try to work out could include:

- **Frozen Tag.** When tagged by a character who is It, a character cannot move until tagged by a character who is not It.
- **Sardines.** One character hides: when another character finds the hiding character, that character joins the hiding place, and play continues until all characters are in the hiding place.
- **Kick the Can.** One character tries to protect a base, and can tag characters who approach it. Tagged characters are frozen unless freed by a character who touches the base (kicks the can). Play continues until all characters are frozen.

Starting Your Own Program

Here are the steps to follow to start writing your own programs from scratch.

1. In the Finder, create a new folder for your project and give it a suitable name. If you plan to use the JEP resource file, JEP resources.rsrc, place a copy of this file in your new project folder. If you want to use any of the JEP units, place these in your folder as well.
2. Launch THINK Pascal, then choose Create from the Project menu and click **New**.
3. Name your project, and save it in your new project folder.
4. Choose **New** from the File Menu and type in your new main program:



5. Choose **Add Window** from the Project menu.
6. Create any units you need and add these to the project. If you want, you can add one or more of the JEP units.
7. If you use JEP's starting main program, change the **uses** clause to include your new units.

8. If you want to use the JEP resource file, add this to your project as well. Choose **Run Options** from the Run menu, click the Resources box, and select JEP resources.rsrc.
9. When you're ready to run your program, choose **Go** from the THINK Pascal Run menu.

Appendix

Technical Reference

Introduction

This reference section provides a brief, functional description of all JEP procedures, functions, types, and global variables used in the GridWalker project.

Some of the functions and procedures, such as `DrawBear`, are used in the early stages of the project and are replaced later by more general routines.

Some of the functions, such as `LegalCell`, are defined slightly differently in the later stages of the project than they are at the beginning. In such cases, the latest definition is described here.

AdjustCoords

Unit: Animation

```
procedure AdjustCoords (VAR PlotArea : rect;  
    VAR SliceArea : rect;  
    Heading : headingType;  
    i : integer);
```

Defines the area of the rectangle that is erased during the animation sequence, based on the specified heading and the phase of the animation cycle. Used by the JEP procedure `AnimateCharacter` to make the animation smoother.

See also: `AnimateCharacter`.

AnimateCharacter

Unit: Animation

```
procedure AnimateCharacter (oldCell : cellType;  
    heading : headingType;  
    icon : integer);
```

Draws a series of four icons starting in the cell specified by `oldCell` and moving in the specified heading. The result gives the appearance of an animated character moving the width of one cell across the screen.

See also: `AdjustCoords`, Icon resource IDs, `headingType`, `cellType`.

appleMenu, etc.

Unit: Globals

```
appleMenu, fileMenu, editMenu, runMenu, speedMenu, strategyMenu,  
  optionMenu : menuHandle;
```

The seven menu handles declared in the Globals unit correspond to menu resources in the resource file JEP Resources.rsrc, which must be attached to any JEP project.

See also: MakeMenus, InMenuBar.

CELLSIZE

Unit: Globals

```
CELLSIZE = 32;
```

The constant CELLSIZE, declared in the Globals Unit, specifies the size of a cell in pixels.

Since icons are 32 by 32 pixels, the setting for CELLSIZE should not be changed unless you also contemplate changing the size of the icons.

See also: MAXROWS, MAXCOLUMNS, GRIDTOP, GRIDLEFT.

cellType

Unit: Globals

```
cellType = record  
    row : integer;  
    col : integer;  
end; {cellType declaration}
```

A record consisting of two integer fields, corresponding to a row and column position in the grid.

See also: NextCell, LegalCell, SameCell.

characterArray

Unit: Globals

```
characterArray : characterArrayType;
```

An array of the type characterArrayType. The elements of this array are of the type characterRecType. The number of elements used at any given time is determined by the global variable totalCharacters.

See also: characterArrayType, characterRecType, totalCharacters.

characterArrayType

Unit: Globals

```
characterArrayType = array[1..MAXCHARACTERS] of characterRecType;
```

A one-dimensional array in which the number of elements is set to the constant MAXCHARACTERS. The elements of this array are records of type characterRecType.

See also: characterArrayType.

characterRecType

Unit: Globals

```
characterRecType = record
    icon : integer;
    currentCell : cellType;
    currentHeading : headingType;
    strategy : strategyType;
end; {characterRecType}
```

A variable of this type is used to keep track of information about a single character. The fields describe the character's icon resource ID, current grid position, current heading, and strategy.

See also: NewCharacterRec, characterArray, characterArrayType.

ClickCell

Unit: Grid Utilities

```
function ClickCell (h: integer;
    v : integer): cellType;
```

Returns the cell position in the grid that contains the horizontal (h) and vertical (v) screen coordinates of the two integer arguments. Uses the global constants GRIDTOP, GRIDLEFT, and CELLSIZE.

See also: cellType.

collision

Unit: Globals

A "project global", defined in the Globals Unit, of type boolean, used to flag a collision between a character and an exit or bomb. This variable is not added until Stage 20.

CreateCharacter

Unit: Array Management

```
procedure CreateCharacter (thisCell: cellType);
```

Draws a character in the specified cell, adds a record of characterRecType to characterArray (in the slot specified by the value of totalCharacters), and updates the record in gridArray corresponding to the specified cell.

CreateCharacter calls the JEP function NewCharacterRec. The character's icon is determined by the value of currentIcon, and its strategy by currentStrategy.

CreateCharacter does not itself increment the value of totalCharacters. This is accomplished by GridClick.

See also: NewCharacterRec, currentStrategy, currentIcon, characterArray.

currentIcon

Unit: Globals

```
currentIcon: integer;
```

A global variable that stores the value of the currently selected palette icon.

See also: Icon resource IDs, InitPalette, DrawPalette.

currentSpeed

Unit: Globals

```
currentSpeed : integer;
```

A global variable that determines the amount of time that elapses between erase and draw cycles in the animation process. The integer value represents ticks (sixtieths of a second).

See also: AnimateCharacter.

currentStrategy

Unit: Globals

```
currentStrategy: strategyType;
```

A global variable that keeps track of the currently selected strategy on the Strategy menu. The value of currentStrategy determines which strategy will be used by a character when it is created.

See also: strategyType, characterRecType, strategyHeading.

DeleteCharacter

Unit: Array Management

```
procedure DeleteCharacter (thisCell: cellType);
```

Deletes the character occupying the specified cell from characterArray, and sets the icon field in the appropriate record of gridArray to 0. Also erases the specified cell. Uses the JEP function SameCell to find the character's record in characterArray.

See also: characterArray, CreateCharacter, GridClick.

DrawBear

Unit: Graphics

```
procedure DrawBear (r, c: integer);
```

Draws the bear icon in the row (r) and column (c) of the grid corresponding to the integer arguments.

See also: EraseBear, DrawCharacter, Icon resource IDs.

DrawCharacter

Unit: Graphics

```
procedure DrawCharacter (thisCell: cellType;
    thisHeading: headingType;
    thisIcon: integer);
```

Draws the character (determined by the integer `thisIcon`) with a heading (`thisHeading`) in the specified cell (`thisCell`).

See also: `cellType`, `headingType`, `DrawBear`, Icon resource IDs.

DrawGrid

Unit: Graphics

```
procedure DrawGrid;
```

Draws a grid in the Drawing window of MAXROWS rows by MAXCOLS columns. Can be used to display the rows and columns of the JEP grid.

See also: MAXROWS, MAXCOLS, CELLSIZE.

DrawKangaroo

Unit: Graphics

```
procedure DrawKangaroo (cell: cellType);
```

Draws the KANGAROO icon in the specified cell.

Note: This procedure is designed for use in the early stages of assembling the GridWalker project. Later in the assembly it is replaced by the more general procedure, `DrawCharacter`.

See also: Icon resource IDs, `cellType`, `DrawCharacter`.

DrawPalette

Unit: Palette Utilities

```
procedure DrawPalette;
```

Draws the palette down the left side of the screen, using the icons defined in `InitPalette`.

See also: `InitPalette`, `DrawPaletteIcon`.

DrawPalettelcon

Unit: Palette Utilities

```
procedure DrawPaletteIcon (paletteRow: integer);
```

Draws a single icon in the palette in the specified row (`paletteRow`). The icon drawn depends on the position of the icon in `paletteArray`. If the icon is the currently selected icon, it is inverted using the QuickDraw procedure `InvertRect`.

See also: `InitPalette`, `DrawPalette`, `currentIcon`.

DropCrumb

Unit: Strategies

```
procedure DropCrumb (thisCell: cellType);
```

Drops an imaginary “crumb” in the specified cell and increments the crumb field in the corresponding record of `gridArray`. Used as part of the `CrumbDropper` strategy.

See also: `CrumbDropperHeading`, `EatCrumb`, `gridCellType`.

EatCrumb

Unit: Strategies

```
procedure EatCrumb (thisCell: cellType);
```

Eats an imaginary “crumb” in the specified cell. Decrements the crumb field in the corresponding record of `gridArray` by 1. Can be used as part of a `CrumbEater` strategy.

See also: `CrumbDropperHeading`, `DropCrumb`, `gridCellType`.

EmptyCell

Unit: Grid Utilities

```
function EmptyCell (thisCell: cellType): boolean;
```

Examines the specified cell (`thisCell`) and returns `true` if the cell is in the grid and unoccupied. Uses the JEP function `InGrid`, and the information stored in the global variable `gridArray`.

See also: `cellType`, `InGrid`, `gridArray`.

EraseBear

Unit: Graphics

```
procedure EraseBear (r, c: integer);
```

Erases the cell in the specified row (`r`) and column (`c`) of the grid.

Note: This procedure can actually be used to erase any cell-- not just one occupied by a bear icon. This procedure is used in the early stages of the assembly process.

See also: `EraseCell`.

EraseCell

Unit: Graphics

```
procedure EraseCell (cell: cellType);
```

Erases the specified cell in the grid. Uses the JEP function `Rectangle` and the `QuickDraw` procedure `EraseRect`.

See also: `cellType`, `Rectangle`, `EraseBear`.

Explode

Unit: Graphics

procedure Explode (thisCell: cellType);

Creates an animated explosion in the specified cell (thisCell) of the grid, which is assumed to be occupied by a bomb. Simultaneously produces a sound using the sound resource with the ID 1000.

See also: cellType, Icon resource IDs.

gameOver

Unit: Globals

gameOver: boolean;

A global variable used to determine when the program ends. When the user chooses **Quit** from the File menu, the variable gameOver is set to true; otherwise it is false. This variable is tested on each pass through the main event loop.

See also: running.

GRID

Unit: Globals

GRID = 2;

The constant GRID is used to label the part of the Drawing window that is occupied by the grid. It is used by the JEP function WindowPart.

See also: WindowPart, PALETTE, NONE.

gridArray

Unit: Globals

gridArray: gridArrayType;

The only variable in the project of the type gridArrayType. The elements of this array record the basic information about the individual cells in the grid, including the icon ID numbers of any characters or obstacles in the grid and the number of “crumbs” in a cell at any given time.

See also: gridArrayType, gridCellType.

gridArrayType

Unit: Globals

gridArrayType = **array**[1..MAXROWS, 1..MAXCOLS] **of** gridCellType;

A two-dimensional array in which the rows and columns of the array are set to the constants MAXROWS (9) and MAXCOLS (14). These constants can be changed for larger screens. The elements of the array are of type gridCellType.

See also: gridArray, gridCellType.

gridCellType

Unit: Globals

```
gridCellType = record
    icon: integer;
    character: integer;
    crumbs: integer;
end;
```

A record variable used to define the elements in the array type `gridArrayType`. It stores information about an individual cell in the grid, including the icon ID number of a character or obstacle, as well as the number of “crumbs” in that cell.

See also: `gridArray`, `crumbDropper`.

GridClick

Unit: Array Management

```
procedure GridClick (thisCell: cellType);
```

The `GridClick` procedure calls one of three other procedures, depending on the contents of the specified cell.

- If a cell is occupied by a character, it deletes that character and decrements the value of `totalCharacters`. If, as a result, there are no characters left in the grid, `GridClick` disables both **Go** and **Stop** on the Run menu.
- If the cell is empty, the current palette icon (`currentIcon`) represents a character, and the value of `totalCharacters` is less than `MAXCHARACTERS`, then `GridClick` calls `CreateCharacter`, increments `totalCharacters`.
- If the currently selected palette icon is an obstacle, or if the user has clicked on an obstacle in the grid, then `GridClick` calls the `MouseDraw` procedure.

See also: `MouseDraw`, `CreateCharacter`, `totalCharacters`.

GRIDLEFT

Unit: Globals

```
GRIDLEFT = 37;
```

The constant `GRIDLEFT` specifies the distance in pixels between the left edge of the screen and the left edge of the grid. When `GRIDLEFT` is set to 37, this leaves enough room for the palette at the left of the screen.

See also: `MAXROWS`, `MAXROWS`, `GRIDTOP`, `CELLSIZE`.

GridSkirterHeading

Unit: Strategies

```
function GridSkirterHeading (thisCell: cellType;
    thisHeading: headingType): headingType;
```

Using a cell (`thisCell`) and a heading (`thisHeading`), returns a heading to a cell in the grid that is not occupied by an obstacle or another character, following this rule: Continue forward if possible; if not, turn right.

See also: `headingType`, `strategyType`, `StrategyHeading`, `RightTurn`.

GRIDTOP

Unit: Globals

```
GRIDTOP = 37;
```

The constant `GRIDTOP`, declared in the `Globals` unit, specifies the distance, in pixels, between the top of the screen and the top edge of the grid. When `GRIDTOP` is set to 37, it leaves just enough room for the menu bar at the top of the screen.

See also: `MAXROWS`, `MAXROWS`, `GRIDLEFT`, `CELLSIZE`.

headingType

Unit: Globals

```
headingType = (west, north, east, south, noHeading);
```

An enumerated type that lists the compass headings. Variables of this type are used throughout construction of `GridWalker`.

See also: `RandomHeading`, `RightTurn`.

Heads

Unit: Strategies

```
function Heads: boolean;
```

Randomly returns `true` or `false`, with an equal probability of either result.





























See also: `LeastCrumbsHeading`.

Icon Resource IDS













Unit: Globals

The icon constants declared in the Globals unit serve to label the resource IDs for the icons in the resource file JEP Resources.rsrc, which must be attached to any JEP project. The icons with their constant identifiers (if any) and resource IDs are given below.

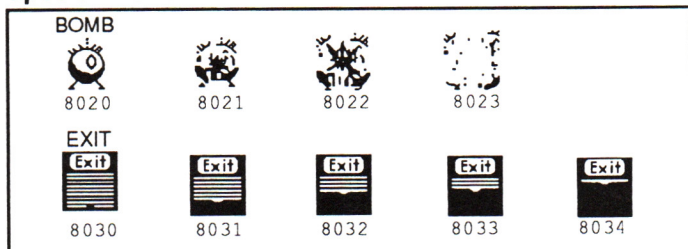
Character Icons

BEAR  3101				BOUNCER  3201			
 3105				 3205			
 3109				 3209			
 3113				 3213			
JEP  3301				TUMBLER  3401			
 3305				 3405			
 3309				 3409			
 3313				 3413			
KANGAROO  3501				ROBOT  3601			
 3505				 3605			
 3509				 3609			
 3513				 3613			
FRED  3701							
 3705							
 3709							
 3713							

Obstacle Icons

HEDGE  8001	FENCE  8002	MAPLE  8003	BOULDER  8004	PINE  8005	ALERT  8006
POISON  8012	DISK  8013	SENSOR  8014	FANPLUG  8015	PODHANGER  8016	CIRCUIT  8017

Special Icons



- Icons representing obstacles (icons that cannot be animated) are given resource IDs in the 8000 range.
- The two special icons are animated using the `MakeExit` and `Explode` procedures.
- Icons representing characters (icons that can be animated) are given resource IDs in the 3000 range. Each character is actually represented by a set of 17 icons. The first icon in the set serves as a palette icon, and the other 16 are used for animation.

See also: `SetBomb`, `SetObstacle`, `DrawCharacter`, `IsCharacter`, `IsObstacle`, `IsBombCell`, `IsExitCell`, `DrawBear`, `DrawKangaroo`, `InitPalette`, `currentIcon`.

InGrid

Unit: Grid Utilities

function `InGrid` (`thisCell`: `cellType`): `boolean`;

Returns `true` if the specified cell is within the grid. Uses the global constants `MAXROWS` and `MAXCOLS`.

See also: `MAXROWS`, `MAXCOLS`, `cellType`.

InitGridArray

Unit: Array Management

procedure `InitGridArray`;

Sets the values of the fields in `gridArray` to 0 and erases the contents of these cells from the screen.

See also: `gridArray`, `gridArrayType`, `gridCellType`.

InitPalette

Unit: Palette Utilities

procedure `InitPalette`;

Sets the values of `paletteArray` to global icon constants such as `BOUNCER`, `SENSOR`, `BOMB`, etc.

Note: You can easily change the contents of the palette simply by changing the values in `paletteArray`.

See also: `DrawPalette`, `DrawPaletteIcon`, icon resource IDs.

InMenuBar

Unit: Menu Utilities

procedure InMenuBar (MyEvent: eventRecord);

Executes a series of case statements dealing with various events related to the menu bar, including selection of items on the File, Run, Speed, and Strategy menus.

See also: currentStrategy, currentSpeed, gameOver.

InPalette

Unit: Palette Utilities

function InPalette (thisCell: cellType): boolean;

Returns true if the specified cell is within the palette; otherwise returns false.

See also: PALETTE.

IsBombCell

Unit: Grid Utilities

function IsBombCell (thisCell: cellType): boolean;

Returns true if the specified cell is occupied by a bomb icon. Uses the information in the global variable gridArray.

See also: Icon resource IDs, Explode.

IsCharacter

Unit: Grid Utilities

function IsCharacter (thisIcon: integer): boolean;

Returns true if the specified icon is in the range 3000-5000, the range of the character icon ID numbers.

See also: Icon resource IDs.

IsExitCell

Unit: Grid Utilities

function IsExitCell (thisCell: cellType): boolean;

Returns true if the specified cell is occupied by an exit icon. Uses the information in the global variable gridArray.

See also: Icon resource IDs, MakeExit, gridArray.

IsObstacle

Unit: Grid Utilities

function IsObstacle (thisIcon: integer): boolean;

Returns `true` if the integer (thisIcon) is greater than 8000, the range of the obstacle icon ID numbers.

See also: Icon resource IDs.

LeastCrumbsHeading

Unit: Strategies

function LeastCrumbsHeading (thisCell: cellType): headingType;

Returns a heading to the cell adjacent to the specified cell (thisCell) that contains the least number of imaginary “crumbs.” In the case of ties, makes a random choice using the JEP procedure Heads.

See also: DropCrumb, StrategyHeading, strategyType.

LegalCell

Unit: Grid Utilities

function LegalCell (thisCell: cellType): boolean;

Returns `true` if the specified cell (thisCell) is within the grid and unoccupied by anything other than a bomb or a exit. Uses the JEP function InGrid and the global variable gridArray.

In the early stages of assembling GridWalker, LegalCell returns `true` as long as the cell is within the grid.

See also: InGrid, IsExit, IsBombCell.

MakeExit

Unit: Graphics

procedure MakeExit (thisCell: cellType;
 thisIcon: integer);

Draws the inverted character icon (based on thisIcon) in the specified cell (thisCell), and then animates the exit closed, while generating a synthesized sound.

See also: PennyWhistle, cellType, Icon resource IDs.

MakeMenus

Unit: Menu Utilities

procedure MakeMenus;

Sets up the GridWalker menu bar, using MENU resources in the file JEP resources.rsrc, which must be attached to any JEP project. The menu bar itself is drawn using the Macintosh toolbox call, DrawMenuBar.

See also: appleMenu.

MAXCHARACTERS

Unit: Globals

```
MAXCHARACTERS = 5;
```

The constant MAXCHARACTERS sets the maximum number of characters allowed in the grid, as opposed to the actual number, which is stored in the global variable totalCharacters. The maximum setting is limited by available memory and the size of the grid.

See also: totalCharacters.

MAXCOLS

Unit: Globals

```
MAXCOLS = 14;
```

The constant MAXCOLS defines the number of columns in the grid. On a standard Macintosh monitor, MAXCOLS should be set no higher than 14.

See also: MAXROWS, GRIDLEFT, GRIDTOP, CELLSIZE.

MAXROWS

Unit: Globals

```
MAXROWS = 9;
```

The constant MAXROWS defines the number of rows in the grid. On a standard Macintosh monitor, MAXROWS should be set no higher than 9.

See also: MAXCOLS, GRIDLEFT, GRIDTOP, CELLSIZE.

MouseDown

Unit: Array Management

```
procedure MouseDraw (thisCell: cellType);
```

Lets a user draw or erase obstacles in the grid with the mouse, starting in the specified cell. If the cell is occupied by an obstacle, MouseDraw erases the cell and continues erasing so long as the mouse button is depressed. If the cell is unoccupied, MouseDraw draws an icon in the cell specified by the global variable currentIcon, and continues drawing as long as the button is depressed.

MouseDown calls various JEP functions and procedures including EmptyCell, Rectangle, InGrid, and IsCharacter. MouseDraw also uses the Toolbox procedure, GetMouse.

See also: GridClick, EmptyCell, InGrid, IsCharacter.

MoveCharacter

Unit: Animation

procedure MoveCharacter;

Locates a new cell for a character in the character array, depending on the current positions of other characters and obstacles in the grid, and the character's assigned strategy.

- If a character moves into a bomb, MoveCharacter calls Explode and sets the collision flag to true.
- If a character moves into an exit, MoveCharacter calls MakeExit and sets the collision flag to true.
- Otherwise, MoveCharacter animates the character into the new cell, assuming one has been found.

See also: MakeExit, Explode, collision.

NewCharacterRec

Unit: Array Management

function NewCharacterRec (thisRow: integer;
 thisCol: integer;
 thisIcon: integer;
 thisStrategy: strategyType): characterRecType;

Returns a record of characterRecType, with fields determined by the arguments to the function. For example, a function call of NewCharacterRec (1, 1, KANGAROO, gridWalker) returns a record for a character represented by the KANGAROO icon, placed in the first row and first column of the grid, and using the gridWalker strategy.

See also: characterRecType, CreateCharacter, strategyType, Icon resource IDs.

NextCell

Unit: Grid Utilities

function NextCell (thisCell: cellType;
 thisHeading: headingType): cellType;

Returns a cellType value that identifies the adjacent cell to the specified cell (thisCell) in the specified heading.

See also: headingType, cellType.

NONE

Unit: Globals

`NONE = 0;`

The constant NONE is used to label the part of the Drawing window that is occupied by neither the GRID nor the PALETTE. It is used by the function JEP function, WindowPart.

See also: WindowPart, GRID, PALETTE.

NumCrumbs

Unit: Strategies

function NumCrumbs (thisCell: cellType): integer;

Returns the number of imaginary “crumbs” in the specified cell. Uses information stored in gridArray.

See also: DropCrumb, EatCrumb, gridArray, LeastCrumbsHeading, gridCellType.

NUMSTRATEGIES

Unit: Globals

`NUMSTRATEGIES = 3;`

The constant NUMSTRATEGIES determines the number of strategies currently defined. It should be equivalent to the number of strategies listed in the enumerated type, strategyType.

See also: strategyType, currentStrategy, strategyHeading.

PALETTE

Unit: Globals

`PALETTE = 1;`

The constant PALETTE is used to label the part of the Drawing window that is occupied by the palette. It is used by the function JEP function, WindowPart.

See also: WindowPart, GRID, NONE.

paletteArray

Unit: Globals

`paletteArray: array[1..9] of integer;`

A one-dimensional array variable in which each element in the array identifies a character or obstacle in the palette. The values stored in the array are determined by InitPalette.

See also: Icon resource IDs, InitPalette, DrawPalette, currentIcon.

PaletteRow

Unit: Palette Utilities

function PaletteRow (thisIcon: integer): integer;

Returns an integer corresponding to the row in the palette, if any, occupied by the specified icon. Returns 0 if the specified icon is not found in paletteArray. PaletteRow is used by the JEP procedure, SwitchPaletteIcon.

See also: SwitchPaletteIcon, paletteArray.

PennyWhistle

Unit: Sound

procedure PennyWhistle (len, startTone, accel: integer);

Produces the sound called by the MakeExit procedure. The integer len determines the length of the sound; startTone sets the pitch that it starts at. The integer accel determines how quickly the pitch changes.

See also: MakeExit.

RandomHeading

Unit: Strategies

function RandomHeading: headingType;

Returns a value of headingType (west, north, east, or south) using the QuickDraw procedure random.

See also: headingType, RandomWalkerHeading, strategyType, StrategyHeading.

RandomMaze

Unit: Array Management

procedure RandomMaze;

Constructs a maze by randomly placing obstacles in the grid. The number of obstacles is determined by MAXROWS. One bomb is placed randomly and one exit is placed in the lower-right corner of the grid.

See also: MAXROWS, MAXCOLS, Icon resource IDs.

RandomWalkerHeading

Unit: Strategies

function RandomWalkerHeading (thisCell: cellType): headingType;

Returns a heading chosen at random to a legal cell adjacent to the specified cell. If no legal cell is available, returns the value noHeading.

See also: headingType, RandomHeading, strategyType, StrategyHeading, LegalCell.

Rectangle

Unit: Graphics

```
function Rectangle (r, c: integer): rect;
```

Returns a value of type `rect` based on the row (`r`) and column (`c`) of a cell in the grid. Used with QuickDraw procedures such as `InvertRect`, which take `rect`-type variables as arguments.

See also: `DrawCharacter`, `DrawBear`.

RightTurn

Unit: Grid Utilities

```
function RightTurn (thisHeading: headingType): headingType;
```

Returns a `headingType` value corresponding to a right turn from the specified heading.

See also: `headingType`, `GridSkirterHeading`.

running

Unit: Globals

```
running: boolean;
```

The global variable is used to determine when to move characters in the grid. The user can choose **Stop** from the Run menu to halt the animation. This sets `running` to `false`.

See also: `gameOver`.

SameCell

Unit: Grid Utilities

```
function SameCell (firstCell: cellType;  
    secondCell: cellType): boolean;
```

Returns `true` if the specified cells have the same grid location; that is, the row and column fields are the same.

See also: `IsBombCell`, `IsExitCell`.

SetBomb

Unit: Graphics

```
procedure SetBomb (r, c: integer);
```

Draws the bomb icon in the cell corresponding to a row (`r`) and column (`c`) in the grid. Is replaced by the more general procedure, `SetObstacle`.

See also: `SetObstacle`, Icon resource IDs.

SetObstacle

Unit: Graphics

```
procedure SetObstacle (thisCell: cellType;  
                       thisIcon: integer);
```

Draws the specified icon (thisIcon) in the specified grid location (thisCell).

See also: Icon resource IDs.

StrategyHeading

Unit: Strategies

```
function StrategyHeading (cell: cellType;  
                           heading: headingType;  
                           strategy: strategyType): headingType;
```

Returns a heading to a legal cell adjacent to the specified cell, following the given strategy and given heading. Returns noHeading if there is no legal move.

See also: strategyType, headingType, cellType.

strategyType

Unit: Globals

```
strategyType = (gridSkirter, randomWalker, crumbDropper);
```

This enumerated type contains the available strategies. If you develop new strategies, you must add their names to this list.

See also: NUMSTRATEGIES, currentStrategy, strategyHeading.

SwitchPalettelcon

Unit: Palette Utilities

```
procedure SwitchPaletteIcon (thisCell: cellType);
```

Changes the currently-selected icon in the palette. The procedure inverts the icon for the currently-selected icon (determined by the value of the global variable currentIcon), inverts the icon in the palette specified by thisCell, and sets the new value of currentIcon.

See also: PaletteRow, paletteArray, currentIcon.

totalCharacters

Unit: Globals

```
totalCharacters: integer;
```

The global variable totalCharacters keeps track of the number of characters in the grid at any one time. It is incremented and decremented by the GridClick procedure.

See also: MAXCHARACTERS.

UpdateGridArray

Unit: Array Management

```
procedure UpdateGridArray (oldCell: cellType;  
    newCell: cellType;  
    icon: integer;)
```

Used to change the information stored in gridArray after a character has moved from one cell in the grid to another. Sets the icon field of the record in gridArray that corresponds to the character's previous cell to 0. Sets the icon field corresponding to the new cell to the value of newCell, assumed to be the character's current cell position.

See also: gridArray, cellType, Icon resource IDs.

Wait

Unit: Graphics

```
procedure Wait (x: integer);
```

Pauses the action for a length of time corresponding to a number of ticks (sixtieths of a second). Uses the Macintosh Toolbox procedure Delay.

See also: currentSpeed.

WindowPart

Unit: Palette Utilities

```
function WindowPart (thisCell: cellType): integer;
```

Returns an integer corresponding to one of three parts of the Drawing window identified by the global constants PALETTE, GRID, or NONE. The result depends on the value of the specified cell. For example, WindowPart (1, 1) returns GRID, WindowPart (1, 0) returns PALETTE, and WindowPart (0, 50) returns NONE (assuming that the value of MAXCOLS is less than 50).

See also: PALETTE, GRID, NONE.

Starting at a Stage

Follow these instructions to begin assembling GridWalker at a particular stage. Use the procedure described here to skip the early stages or to recover from an error in the assembly process.

Copy and Rename a Stage Folder

1. Choose **Quit** from the File menu to leave THINK Pascal and return to the Finder.
2. Open your backup copy of the folder called **Assembly Stages** that was supplied on the JEP disk.
3. Duplicate the folder called **Assembly Core** and give it an original name, like **GW Folder**. If you are working on a two-drive system, copy this new folder onto your work disk.

Copy the Needed Documents

4. Open the folder containing the project completed through the previous stage. For example, if you want to begin with Stage Twelve, open the folder **Stage 11 Complete**.
5. With this folder open, choose **Select All** from the Edit menu to select **all** documents in the **Stage 11 Complete** folder. Hold the Option key down and drag these documents into the folder you just created, **GW Folder**.
6. Click "Yes" when you get the message, "Replace items with the same name with the selected items?"

Caution: Make sure you are copying in the right direction—from the Completed Through... folder into the folder you created. Holding the Option key down while dragging the documents will automatically create duplicates, leaving the original folder unchanged.

Run THINK Pascal, Open the Project, and Add Units

7. Launch THINK Pascal. Choose **Open Project** from the Project menu. Open the **GW Folder** and then open the project in that folder, **GridWalker Project**.
8. Choose **Add File** from the Project menu and one-by-one select the files that belong in the project at that stage. The files you need and the order they belong in are listed on the next page. Move these files so that they are in the correct build order.

Break Into Segments

9. If you are assembling from Stage Ten or later, you must also break the project into two segments. Click the pyramid icon in the project window and then drag unit names below the segment line. Detailed instructions are given in the Instructions window for Stage Eleven Assembly.

Run the New Project

10. Choose **Go** from the Run menu to run the project. Locate the **Runtime.lib**, **Interface.lib**, libraries and the units, following the procedure described in the *Introduction and Installation* section.

To Recover From an Assembly Error

If you make a mistake during the assembly and you want to use the Assembly Stages documents to reconstruct your project, follow this procedure:

1. In the Finder, open the **Stage nn Complete** folder for the previous stage and select all of the documents in it. If you want to start with Stage Eleven, open the folder **Stage 10 Complete**.
2. Drag these documents into **your** GridWalker folder. Click "Yes" to the message.
3. Check the Build Order below. Break into segments if necessary.
4. Continue assembling from the next stage.

Build Order at Each Assembly Stage

Stages One to Four Complete

- Globals Unit
- Sound Unit
- Graphics Unit
- Main Program

Stages Five to Seven Complete

- Globals Unit
- Sound Unit
- Grid Utilities Unit
- Graphics Unit
- Main Program

Stage Eight Complete

- Globals Unit
- Sound Unit
- Grid Utilities Unit
- Graphics Unit
- Strategies Unit
- Main Program

Stages Nine and Ten Complete

- Globals Unit
- Sound Unit
- Grid Utilities Unit
- Graphics Unit
- Strategies Unit
- Array Management Unit
- Main Program

Stage Eleven Complete

- Globals Unit
- Sound Unit
- Grid Utilities Unit
- Graphics Unit
- Strategies Unit
- Array Management Unit
- Animation Unit
- Main Program

Stages Twelve to Sixteen Complete

- Globals Unit
- Sound Unit
- Grid Utilities Unit
- Graphics Unit
- Strategies Unit
- Array Management Unit
- Animation Unit
- Palette Utilities Unit
- Main Program

Stages Seventeen to Twenty Complete

- Globals Unit
- Sound Unit
- Grid Utilities Unit
- Graphics Unit
- Strategies Unit
- Array Management Unit
- Animation Unit
- Palette Utilities Unit
- Menu Utilities Unit
- Main Program

Index

- AdjustCoords 106, 165
- algorithms 70
- and** 53
- AnimateCharacter 104, 165
- animation sequence 41
- Animation unit 104
- AppendMenu 151
- appleMenu 166
- arguments 25
- array 86
- Array Management 89
- Assembly Stages
 - folder 5
- assignment statements 27
- begin** 24
- boolean
 - variable type 57
- build application 161
- build order 10, 56
- case** 71
- case** statement 82
- cell 45
- CELLSIZE 126, 166
- cellType 46, 50, 166
- character icons 104, 174
- characterArray 96, 166
- characterArrayType 96, 167
- characterRecType 64, 167
- CheckItem 152
- ClickCell 125, 167
- collision 161, 167
- comments 68
- compiler 7
- computer program
 - overview 29
- const** 41
- constants 41
- CreateCharacter 139, 167
- CrumbDropper 154
- currentIcon 139, 142, 168
- currentSpeed 107, 150, 168
- currentStrategy 138-139, 142, 168
- declaring variables 26
- Delay 107
- DeleteCharacter 139, 168
- DisableItem 146
- div** 53, 83
- documents 8
- DrawBear 27, 40, 43, 169
- DrawCharacter 169
- DrawGrid 169
- Drawing window 25
 - making larger 113
- DrawKangaroo 169
- DrawPalette 113, 169
- DrawPaletteIcon 112, 170
- DropCrumb 154, 157, 170
- EatCrumb 157, 170
- edit window 14
- editor
 - in THINK Pascal 7
- else** 91
- EmptyCell 126, 170
- EnableItem 146
- enumerated type 65, 78
- EraseBear 38, 170
- EraseCell 171
- EraseRect 40
- evaluation
 - of an expression 33
- eventMask 118
- eventRecord 118
- exit test
 - in a **repeat** loop 34
- Explode 52, 171
- expression 27
 - in assignment statement 33
 - boolean 34
- fields
 - of a record 47
- FindWindow 120

- for** loop 52
 - nested 92
- FrameRect 112
- functions 58
 - in an expression 60
- gameOver 57, 171
- GetNextEvent 119
- Globals unit 41
- Go-Go command 68
- GRID 171
- gridArray 88, 171
- gridArrayType 87, 172
- gridCellType 87, 172
- GridClick 134, 140, 172
- GRIDLEFT 126, 173
- GridSkirterHeading 81, 173
- GRIDTOP 126, 173
- GridWalker 2
 - folder 5
- handle 144
- headingType 65, 173
- Heads 156, 173
- icon resource IDS 174
- icons
 - about 42
- identifiers 35
 - uppercase vs. lowercase 66
- if** statement 53
- InGrid 175
- InitGridArray 89, 175
- InitPalette 111, 175
- InMenuBar 176
- InPalette 176
- InsertMenu 145, 146
- Instant window 54
- integer 26
- interface
 - section of a unit 38
- Interface.lib 40
- IsBombCell 90, 176
- IsCharacter 124, 176
- IsExitCell 90, 176
- IsObstacle 125, 177
- iteration 32
- JEP palette 110

- LeastCrumbsHeading 155, 177
- LeftTurn 75
- LegalCell 58, 91, 177
- Library documents 9
- LightsBug 93
- logical operators 53
- longint 146
- loops
 - for** loop 52
 - repeat** loop 31
 - while** loop 155
- main event loop 116
- main program 13
- MakeExit 75, 177
- MakeMenus 144, 177
- MAXCHARACTERS 96, 178
- MAXCOLS 41, 178
- MAXROWS 178
- menu bar 144
- menuHandle 144
- MenuSelect 146
- mod** 53, 83
- mousedown event. 116
- MouseDown 132, 178
- MoveCharacter 122, 161, 179
- nesting 92
- NewCharacterRec 97, 179
- NextCell 71, 179
- NONE 179
- not** 53
- NumCrumbs 155, 180
- NUMSTRATEGIES 180
- Observe window 35
- obstacle icons 174
- operators
 - logical 53
- options
 - debugging 10
- or** 53
- ord** 106
- otherwise**
 - in a **case** statement 71
- palette 110
- PALETTE
 - constant 180

- Palette Utilities unit 111
- paletteArray 111, 180
- PaletteRow 181
- parameter list
 - in procedure definition 39
- parameters 25
 - actual 39
 - formal 39
- PennyWhistle 181
- precedence 53
- pred 71
- procedure call 25
- procedure definition
 - interface part 39
- procedures 25, 40
- program block 24
- program header 24
- project 8
- project document 8
- QuickDraw
 - experiments with 113
- Random 83, 84, 156
- random number generator
 - seeding 84
- RandomHeading 181
- RandomMaze 93, 135, 181
- RandomWalkerHeading 78, 181
- randSeed 84
- record
 - assigning values to 65
 - in a procedure call 49
- record type 46
- rect 40, 113
- Rectangle 182
- repeat** loop 31, 37
- Reset Command 68
- ResetRunMenu 148
- resource file 42, 43
- RightTurn 70, 182
- Run Options 43
- running 182
- Runtime.lib 40
- SameCell 60, 182
- selector variable
 - in a **case** statement 71
- SetBomb 25, 75, 182
- SetObstacle 75, 89, 183
- SetRect 112
- setting flags 161
- ShowDrawing 25, 40
- source code 68
- special icons 175
- Step Command 30
- Step Into Calls Command 43, 75
- strategy 75
- strategyHeading 82, 183
- strategyType 78, 183
- string 146
- succ 71
- SwitchPaletteIcon 128, 183
- SysBeep 142
- TickCount 84
- totalCharacters 97, 99, 138, 183
- Trace command 61
- type
 - predefined 46
 - record type 47
 - Toolbox types 46
 - type declaration 46
 - user-defined 46
- units 10
- until** 32
- UpdateGridArray 100, 184
- user interface 110
- uses** clause 24, 57
- values
 - assigning to a record variable 48
 - assigning to variables 26
- variable declaration 24
- variables 26
 - declaring 47
 - global vs. local 57
 - local 52
 - passing to procedures 27
 - types 26
- View Options 61
- Wait 40, 184
- what
 - field of event record 118
- while** loop 155

WindowPart 127, 184
with 66, 97

License Agreement

License Agreement

This manual and the software described in it were developed and are copyrighted by Symantec Corp. (Symantec) and are licensed to you on a non-exclusive, non-transferable basis. Neither the manual nor the software may be copied in whole or in part except that you may make backup copies of the software for your use provided that they bear Symantec's copyright notice.

You may not in any event distribute any of the source files provided or licensed as part of the software. You may use the software at any number of locations so long as there is no possibility of it being used at more than one location at a time.

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed or in the manuals distributed with the software, Symantec will replace the media or manuals at no cost to you provided that you return the defective materials along with a copy of your receipt to Symantec or to an authorized Symantec dealer during the 60-day period following your receipt of the software.

Limited Warranty on the Product

Symantec warrants that the software will perform substantially as described in the User's Manual. If within 60 days of receiving the software, you give written notification to Symantec of a significant, reproducible error in the software which prevents operation, and provide a written description of the possible problem along with a machine readable example, if appropriate, Symantec will either provide you with corrective or workaround instructions, a corrected copy of the software, a correction to the User's Guide and Reference Manual, or Symantec will refund your purchase price upon return of all copies of the software and documentation together with a copy of your receipt. This warranty extends only to you and shall be void if the software has been tampered with, modified, or improperly used, or if the software is used on hardware other than the Apple Macintosh™ Computer.

EXCEPT FOR THE LIMITED WARRANTY DESCRIBED ABOVE, THERE ARE NO WARRANTIES TO YOU OR ANY OTHER PERSON OR ENTITY FOR THE PRODUCT EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. Some states do not allow the exclusion of implied warranties or limitations on how long they last, and you also may have other rights that vary from state to state. IN NO EVENT SHALL SYMANTEC BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO YOU OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE

LEGAL THEORY, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. The warranty and remedies set forth are exclusive and in lieu of all others, oral or written, express or implied.

30 Day Money-Back Guarantee

If within 30 days of purchase this product does not perform in accordance with our claims, you may return it to your dealer for a refund. If you purchased it directly from Symantec Corporation, a Returned Merchandise Authorization (RMA) number must be assigned. To obtain an RMA number, please call (408) 253-9600 between 8:00 a.m. and 5:00 p.m. Pacific time. You will be given an RMA number and instructions on where to return the product. No returns will be accepted without an RMA number.